TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií

Studijní program: B2646 – Informační technologie
Studijní obor: 1802R007 – Informační technologie

Systém pro podporu navrhování optimální struktury softwarových projektů

Framework Supporting Proper and Optimal Design of Software Products

Bakalářská práce

Autor: **Martin Tomáš**
Vedoucí práce: Ing. Roman Špánek, Ph.D.
Konzultant: Ing. Pavel Tyl

V Liberci 8. 5. 2012

Vložení originálního zadání.

Prohlášení

Byl jsem seznámen s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu TUL.

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Bakalářskou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím bakalářské práce a konzultantem.

Datum

Podpis

Abstrakt

Bakalářská práce si klade za cíl nastínit řešení automatizovaného návrhu optimální struktury objektově orientovaného problému na základě jeho definice. Popisuje metody možného definování problému a vysvětluje jejich vhodnost pro navrhovaný automatizovaný systém. Za pomoci znalosti těchto poznatků byl zvolen vizuální jazyk UML a jeho diagram tříd pro znázornění problému. Optimální řešení je vybráno na základě shody mezi již vyřešeným problémem a aktuálním řešeným problémem. Shoda je vyjádřena největším podgrafem nalezeným pomocí grafových algoritmů na příslušných diagramech.
Navržená aplikace dovoluje uživateli spravovat vlastní již vytvořená optimální řešení daného problému a ty použít jako vzory pro vyřešení aktuálního problému. Uložené diagramy jsou převedeny na graf a za pomoci teorie grafů mezi sebou vzájemně porovnávány. Posléze je uživateli předložena možnost zvolit právě takové řešení daného problému, které nejvíce odpovídá nalezené shodě.

Abstract

This Bachelor's Thesis aims to outline the solution of automated design of optimal structure of object-oriented problem based on its definition. It describes methods of how to define a problem and explains their suitability for the proposed automated system. UML language, particularly the class diagram, was selected for input description of the problem for its popularity,extensibility and standardization. Selection of the optimal solution (program structure) is based on mining for a similarity in a knowledge base created from users' former project designs. The similarity is formulated on the largest subgraph found in diagrams with using graph algorithms.
The designed application allows users to manage their own solutions of the problems and use them as patterns for current problems. Stored diagrams are converted to graphs and consequently compared and contrasted to themselves by using graph theory. Users are presented with the possibility to opt for such a solution of the problem that matches the similarity the most.

Obsah

Seznam obrázků

Seznam zkratek

BFS procházení do šířky

DFS procházení do hloubky

GoF Gang čtyř

GUI grafické uživatelské rozhraní

MCES maximální společný hranový podgraf

MCIS maximální společný indukovaný podgraf

MCS maximální společný podgraf

MVC model-pohled-kontrolor

OO objektově orientované

OOP objektově orientované programování

POAD analýza a design orientovaný na vzory

UML unifikovaný modelovací jazyk

UP unifikovaný proces

1 Úvod

Cílem bakalářské práce bylo rozvinutí znalostí ohledně návrhových vzorů a objektově orientovaného programování (oop). Práce se soustředila na metody výběru vhodné struktury na základě jeho definice a na návrh systému, který by dovoloval takovýto výběr učinit. Pro návrh výsledného systému byl použit složený návrhový vzor MVC.
Výstupem bakalářské práce je aplikace dovolující uživateli spravovat už navržené struktury projektů a na nich potom vyhledávat optimálnější řešení pomocí diagramů tříd jazyka UML. Na základě takto definovaného problému a již existujících optimálnějších řešení lze vybrat další vhodný postup návrhu softwaru. Tím se výrazně zvyšuje znovupoužitelnost již navržených programů a snižuje náročnost zvolení vhodného řešení nebo návrhového vzoru pro budoucí projekt.

2 Proces vývoje softwaru

Vývoj programu je poměrně komplexní činnost, a proto jsou u rozsáhlejších projektů vždy zapotřebí služby analytika, designéra a dalších, aby bylo možno tvořit a pokračovat v projektu nejen rychleji, ale také méně nákladně, popřípadě aby jeden programátor dokázal zastoupit druhého. Neméně důležitým aspektem vývoje softwaru je jeho budoucí snadná rozšířitelnost a náročnost dalších úprav.
Vývoj softwaru se skládá z několika etap, které se mohou překrývat. Nemělo by ovšem docházet k vynechání jedné či více z nich. Tyto etapy mohou být realizovány vodopádovým modelem (jedna po druhé, postupně) nebo přírůstkovou metodou (vytváříme vždy část aplikace, přírůstek), případně další, vhodnější metodikou.

1. požadavky – shrnutí všech požadavků na software
2. analýza – definování struktury požadavků
3. design – realizace požadavků v systému
4. implementace – budování programu
5. test – testování aplikace (výstup odpovídá zadaným požadavkům)

K vytvoření takovéto dokumentace může sloužit **UML** (Unified Modeling Language), který dokáže znázornit a vymodelovat převážnou část aplikace. Díky tomu neztratíme orientaci ani ve velkých, rozsáhlých projektech.
Návrhové vzory potom dovolují téměř v počátcích vývoje použít ověřené řešení, a tak se v budoucnu vyhnout případným komplikacím. Tím výrazně zlevňují, zrychlují a zkvalitňují celý proces.

3 Objektově orientované programování

Objektově orientované programování dovoluje rozdělit velmi složité funkční celky na jednodušší jednotky. Takový přístup umožňuje paralelní vývoj softwaru, zjednodušit budoucí úpravu navržených jednotek a jejich znovupoužití.

3.1 Principy

Objektově orientované programování využívá v zasadě několik principů.

- zapouzdření–k atributům dané třídy přistupujeme pouze prostřednictvím metod

- dědičnost–třída může být potomkem jiné třídy, v tomto případě sdílí stejné vlastnosti rozšířené o své vlastnosti (pro návrh předků využíváme abstrakci, omezení redundance a využití polymorfismu)
- polymorfismus–objekt mění své chování podle instance třídy (pokud má objekt společné rozhraní popřípadě předka s jiným objektem, potom polymorfismus dovoluje pracovat s těmito objekty jednotně, i když jejich chování se liší)
- třída–jedná se o abstraktní definici určitého objektu
- skládání tříd–jedna třída může být složená i z jiných tříd a využívat její metody a atributy (pokud to umožňují práva dané třídy)

Správně napsaný objektově orientovaný program se snaží maximálně využívat těchto principů.
Některé další principy, které se uplatní pří návrhu objektově orientované aplikace mohou být třeba: každá třída by měla mít minimální počet závislostí, měla by dávat přednost skládání tříd oproti dědičnosti, snažit se o volné spojení mezi třídami, třídy by jsme měli navrhovat oproti rozhraním a mnoho dalších.[1]

4 UML

UML je dnes jeden z nejrozšířenějších vizuálních modelovacích jazyků pro návrh systémů a programů. Návrhové vzory jsou často popisovány a znázorňovány právě pomocí tohoto jazyka a jeho diagramů, kvůli zachování přenositelnosti a vysoké míry abstrakce.[3]

4.1 Princip UML

Pomocí UML lze vymodelovat celý životní cyklus softwaru, od jeho prvopočátků (definování prvotních požadavků) až po finální část. UML je pak tvořeno bloky (elementy, vztahy mezi nimi a diagramy), společnými mechanismy a architekturou systému.
Každý vytvořený model nebo spojení mezi nimi je přidáno do diagramu, avšak **diagram sám o sobě není modelem**. Diagram pouze poskytuje pohled na model, proto i když smažeme model z diagramu, sám model bude stále existovat.
Každý z diagramů poskytuje vlastní znázornění části systému, ať už jeho toku, a nebo struktury. Příkladem návrhu jednoduššího systému potom může být společné použití například těchto diagramů:

- diagram případu užití (UseCase) pro návrh vstupujících a vystupujících uživatelských rolí systému
- diagram tříd (Class) pro návrh odpovídajících tříd
- sekvenční (Sequence) diagram pro znázornění komunikace mezi těmito třídami
- stavový (State) diagram pro ukázku vnitřních stavů složitějších tříd
- použití diagramů složení a balíčku (Composite a Package) pro zpřehlednění a funkční rozdělení aplikace

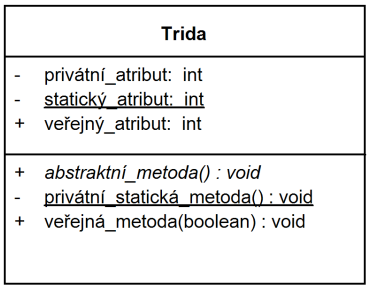
Samozřejmostí jsou i jednotlivé druhy vazeb, které nám dovolují propojovat určené modely na daných diagramech mezi sebou a tím vytvářet složitější struktury[2].

4.2 Objektové programování a UML

UML je ideální nástroj pro objektově orientované modelování, protože poskytuje dokonalé možnosti znázornění nejen vzájemné komunikace jednotlivých objektů, ale i vnitřních toků celé aplikace a jednotlivých tříd.[10]

4.2.1 Diagram tříd

Mnou vytvořená aplikace využívá diagram tříd pro porovnávání dvou struktur, z toho důvodu ho na tomto místě popíšu detailněji. Element třídy dovoluje přehledně definovat jméno třídy, atributy, metody a jejich vlastnosti. Jednotlivé možnosti znázornění vlastností třídy ukazuje přehledně Obrázek 1. Jednotlivé elementy můžeme spojovat pomocí různých vazeb (např. asociace, dědičnost, atd.).



Obrázek 1: UML element třídy

5 Návrhové vzory

Každý systém, i třeba poměrně jednoduchý, má tendence, pokud je správně navržen, vytvářet **stejné struktury tříd a vazeb**, které jsou řešením určitých stále stejných problémů. Pokud je člověk zkušenějším programátorem, dokáže tyto struktury odhalit a efektivně znovu použít. Snaha ukázat tyto postupy vedla k jejich standardizaci a vytvoření rejstříků návrhových vzorů.[6]

5.1 Anti-návrhové vzory

Společně se vznikem návrhových vzorů, které popisovali správný postup řešení problému, vznikli také anti-vzory ukazující, jak by se daný problém v žádném případě řešit neměl.

5.2 Typy návrhových vzorů

Návrhových vzorů existuje celá řada. Pokud si jednotlivý vzor představíme jako recept v kuchařce, potom takovýchto kuchařek budeme mít celou řadu, jednu pro databáze, jednu pro objektově orientované programování a jiné další. I přes obrovské množství těchto vzorů je užitečné umět je používat a pochopit jejich základní parametry a principy. Pokud si osvojíme tyto znalosti, budeme moci ke každému problému efektivně najít odpovídající vzor a tím i řešení.[5]

5.2.1 GoF vzory

Zaměříme se nyní na objektově orientované programování a GoF (gang of four) vzory. Jedná se o návrhové vzory poskytující řešení nejčastějších problémů spojených s objekto-

vým programováním, prací s objekty a jejich vhodným uspořádáním podle nejvhodnějších objektových principů.[4]

1. Creational vzory (Tvořivé vzory)–vzory související s vytvářením objektů
    - Abstract Factory (Abstraktní továrna), Builder (Stavitel), Factory Method (Tovární metoda), Prototype (Prototyp), Singleton (Jedináček)
2. Structural vzory (Strukturální vzory)–vzory zabývající se tím, jak jsou třídy a objekty poskládány a jak formovat složitější struktury; definují také, jak skládat objekty, a tím jim přidávat nové vlastnosti
    - Adapter (Adaptér), Bridge (Most), Composite (Strom), Decorator (Dekorátor), Facade (Fasáda), Flyweight (Muší váha), Proxy (Zástupce)
3. Behavioral vzory (vzory Chování)–tyto vzory se zabývají algoritmy, komunikací, předáváním úkolů a povinnosti mezi objekty a třídami
    - Chain of Responsibility (Zřetězení zodpovědnosti), Command (Příkaz), Interpreter (Interpret), Iterator (Iterátor), Mediator (Prostředník), Memento (Memento), Observer (Pozorovatel), State (Stav), Strategy (Strategie), Visitor (Návštěvník), Template Method (Šablonová metoda)

Těchto návrhových vzorů je celkem 23, a každý je řešením na určitý problém. V reálných systémech existují tyto vzory ne vždy exaktně aplikované, ale jejich interpretace je občas částečně upravena, aby vzor co nejlépe odpovídal požadavkům systému. Dále pak existují složené vzory obsahující dva a více těchto návrhových vzorů (**Složené vzory**), příkladem takového vzoru je třeba MVC (Model View Controller).

5.3 Popis vzoru

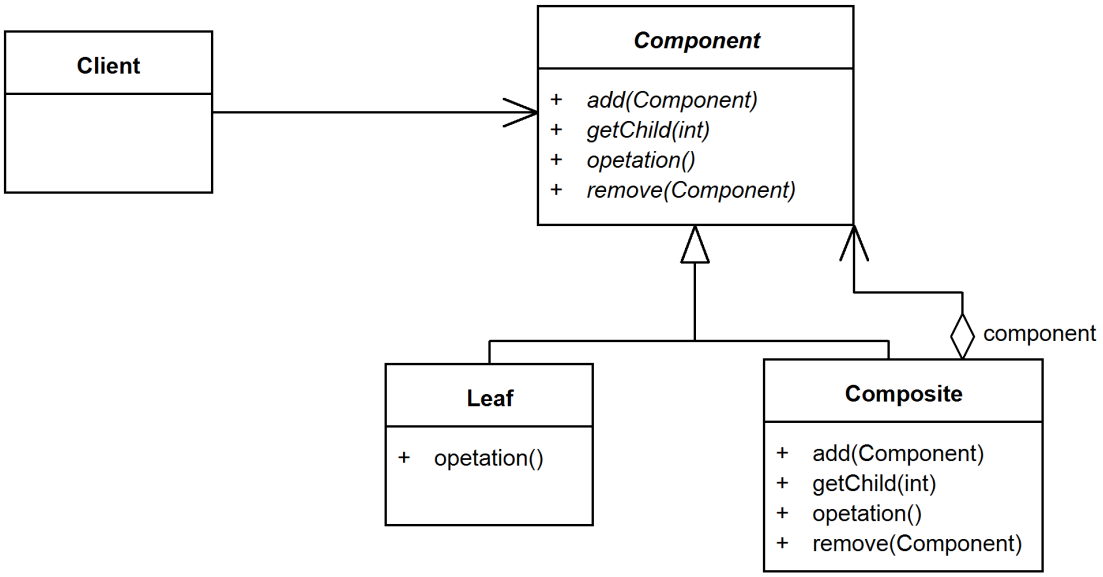
Každý návrhový vzor je definován svým vlastním originálním **jménem**. Unikátní jméno je velmi důležité, protože usnadňuje práci nejen v týmu, ale i mezi týmy. Jeden z nejpodstatnějších úkonů při definici vzoru je přesné **určení problému nebo problémů**, který daný vzor řeší. Dále pak vzor obsahuje popis vhodné implementace, vysvětlení použitého řešení a UML diagram tříd usnadňující pochopení a použití návrhového vzoru.

5.4 Bližší charakteristiky několika vybraných GoF vzorů

Vzhledem k rozsahu a charakteristice práce zde zmíním pouze několik vybraných GoF vzorů. Tyto vzory byli použity při návrhu výsledné aplikace a nebo jsou pro ni jiným způsobem důležité. Můžeme zde také vidět jak definovat daný problém tak, aby byl v budoucnu použitelný jako návrhový vzor.
Z důvodu standardizace jsou zde uvedené UML diagramy v angličtině.

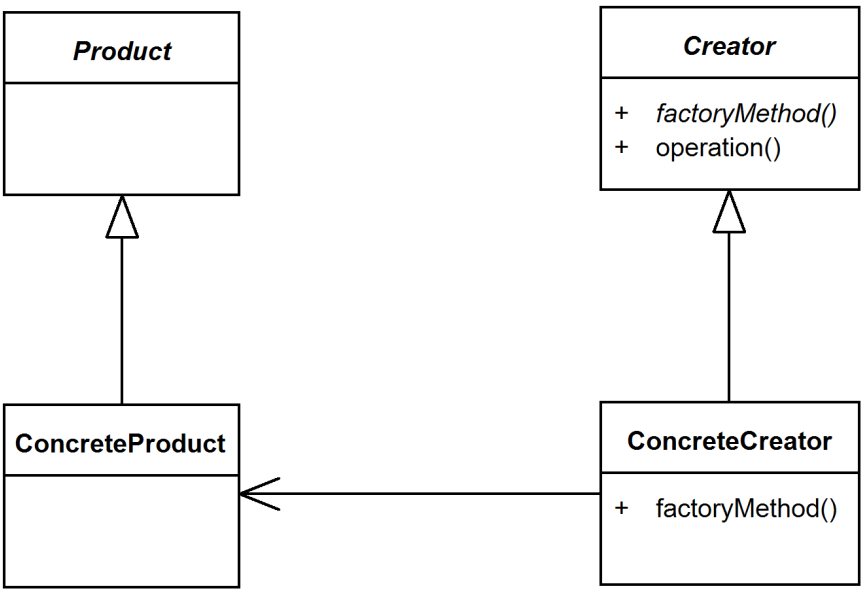
5.4.1 Composite (Strom)

- **Popis:** Composite návrhový vzor patří do skupiny Strukturálních návrhových vzorů, zajímá se tedy o strukturu programu. Poskytuje řešení pro bezpečný návrh stromové struktury tříd, která může obsahovat jak jednoduché objekty (leafs), tak objekty složené (composite). Z těchto objektů potom dovoluje složit stromovou strukturu, tak aby odpovídala požadované hierarchii aplikace. Velkou výhodou tohoto vzoru je, že dovoluje přistupovat jednotně jak k objektům, tak ke složeným objektům.



Obrázek 2: Composite (Strom)

- **Definice:** Composite návrhový vzor dovoluje složit jednotlivé objekty do stromové struktury, reprezentující tak celkovou hierarchii. Tento vzor dovoluje klientovi nakládat s objekty a složenými objekty jednotně.
- **Použití:**Tento vzor je tvořen jednou abstraktní třídou (případně rozhraním–záleží na programovacím jazyce), která obsahuje společné metody jak pro složené objekty (reprezentovány Composite třídou), tak pro jednoduché objekty (reprezentovány Leaf třídou). Tato abstraktní třída je předkem už dvou zmíněných tříd a dovoluje nám tedy přistupovat stejně k obou typům objektů. Při deklaraci všech metod v tomto předkovi vzniká situace, kdy některé metody nemají smysl pro jednoduché objekty, jiné pro složené, je proto nutné tyto stavy ošetřit. Tento problém lze řešit i tak, že abstraktní třída bude obsahovat pouze společné metody a specializované metody budou definovány pouze ve třídách, kam patří. Nemusíme tedy ošetřovat situace, kdy jedna třída je požádána provést akci, která ji nepřísluší.
  Příkladem použití tohoto velmi užitečného vzoru může být třeba grafické rozhraní. Základní okno obsahuje různé panely, ty potom další různá tlačítka, atd. (vidíme zde stromovou strukturu).
- **Základní vlastnosti ve zkratce:**
  1. strukturální
  2. dovoluje vytvářet stromovou strukturu obsahující složené objekty a jednoduché objekty
  3. k nim potom poskytuje jednotný přístup (dosaženo společným předkem)
  4. společný předek může obsahovat všechny metody (musíme ošetřit to, že některé metody jsou použitelné jen jedním typem třídy), a nebo jen společné metody (třídy si své specializované metody definují samy)
  5. časté použití s návrhovým vzorem Iterátor – dovoluje jednotné procházení všech jednoduchých i složených objektů



Obrázek 3: Factory method (Tovární metoda)

5.4.2 Factory method (Tovární metoda)

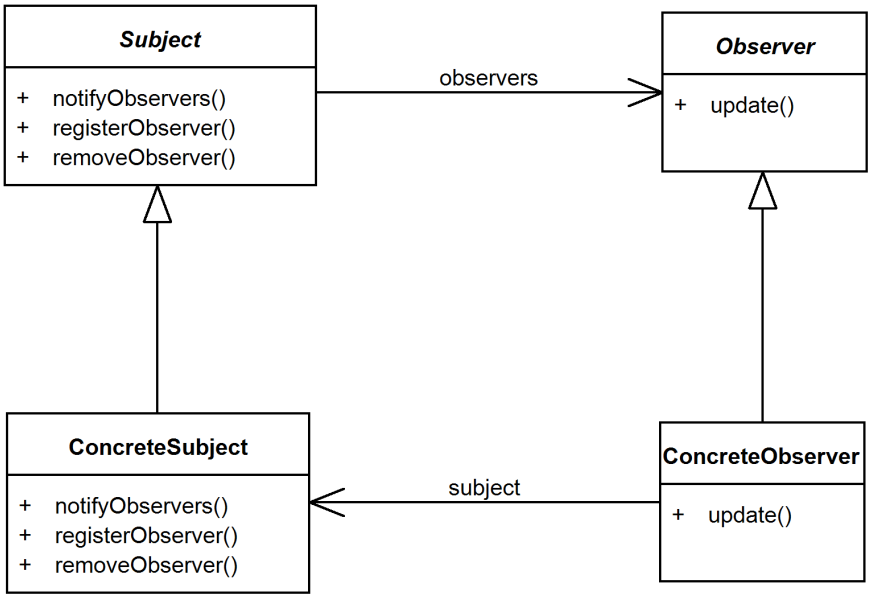
- **Základní informace a vlastnosti:** Návrhový vzor Tovární metody patří mezi Vytvářecí vzory, zajímá se tedy o metodu vytváření objektů. Každá objektově orientovaná aplikace potřebuje nějak vytvářet objekty. Tento vzor poskytuje postup a návrh řešení, jak samotné vytváření vhodně zapouzdřit.

- **Definice:** Tento návrhový vzor definuje rozhraní pro vytváření objektů a také dovoluje podtřídě rozhodnout o druhu vytvářeného objektu. Vzor Tovární metody dovoluje třídě pozdržet vytvoření objektu.

- **Použití:** Princip tohoto vzoru spočívá v tom, že oddělíme vytváření objektu (zapouzdříme ho) do oddělené třídy. Tato třída je potomkem abstraktní třídy, sdružující všechny třídy Tovární metody. Samotný klient říká objektu Tovární metody, jaký objekt si přeje vytvořit, a tak pouze objekt Tovární metody má všechny znalosti o tom, jaký objekt byl konkrétně vytvořen a za jakých podmínek. Zde je využit polymorfismus. Tímto postupem dochází k úplnému zapouzdření samotného vytváření objektu. Vzor Tovární metody také odděluje veškeré závislosti mezi klientem a další částí programu.

- **Základní vlastnosti ve zkratce:**

  1. vytvářecí
  2. odděluje vytváření jednotlivých objektů od klienta, ten inicializuje pouze objekt Tovární metody
  3. odděluje závislost mezi klientem a programem
  4. dochází k zapouzdření vytváření objektů
  5. veškeré postupy a údaje nutné pro vytváření objektu jsou uchovány mimo samotného klienta, tedy v objektu Tovární metody



Obrázek 4: Observer (Pozorovatel)

5.4.3 Observer (Pozorovatel)

- **Základní informace a vlastnosti:** Návrhový vzor Pozorovatel patří do skupiny vzorů Chování, zajímá se tedy o chování systému. Tento návrhový vzor vytváří dvě rozhraní. Subject (Předmět), který definuje základní vlastnosti vzoru, a Pozorovatele, který slouží pro definici společných částí všech posluchačů. Vlastností tohoto vzoru je, že vytváří volné spojení mezi jedním předmětem a jedním nebo několika posluchači.

- **Definice:** Návrhový vzor Pozorovatel definuje jedno až několik závislostí mezi objekty, a to tak, že pokud jeden objekt změní svůj stav, všechny ostatní objekty jsou upozorněny a pozměněny automaticky.

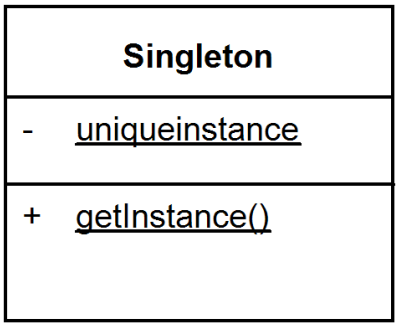
- **Použití:** Použití tohoto vzoru je možné v mnoha aplikacích a samotný princip a účel tohoto návrhu je velmi jednoduchý. Pro ukázku si můžeme představit jeden zdroj dat, třeba uživatelské rozhraní, tedy samotného uživatele. Vytvoříme proto odpovídající objekt Předmětu, ve kterém si budeme udržovat seznam všech dalších objektů, které tento objekt, tedy uživatel ovlivňuje. Mezi objekty vzniká volné spojení (objekty jsou sice spojeny, ale udržují si velmi málo vědomostí jedna o druhé–toho je dosaženo obecnými rozhraními tříd), jedna ku několika (jeden předmět a několik posluchačů). Kdykoliv potom dojde k požadované změně v objektu Předmětu, jsou automaticky upozorněny všichni posluchači a jejich data jsou patřičně upravena, a to buď tak, že Předmět sám zavolá jejich metodu, například update a oni podle toho zareagují, a nebo sám podstrčí všem objektům nově získaná data.
  Příkladem použití návrhového vzoru Posluchače je třeba v jazyce Java, GUI a listenery jednotlivých objektů.

- **Základní vlastnosti ve zkratce:**

  1. vzor Chování
  2. mezi Předmětem a ostatními třídami je volné spojení
  3. posluchači jsou automaticky upravovány, resp. kontaktovány v závislosti na změně v objektu třídy

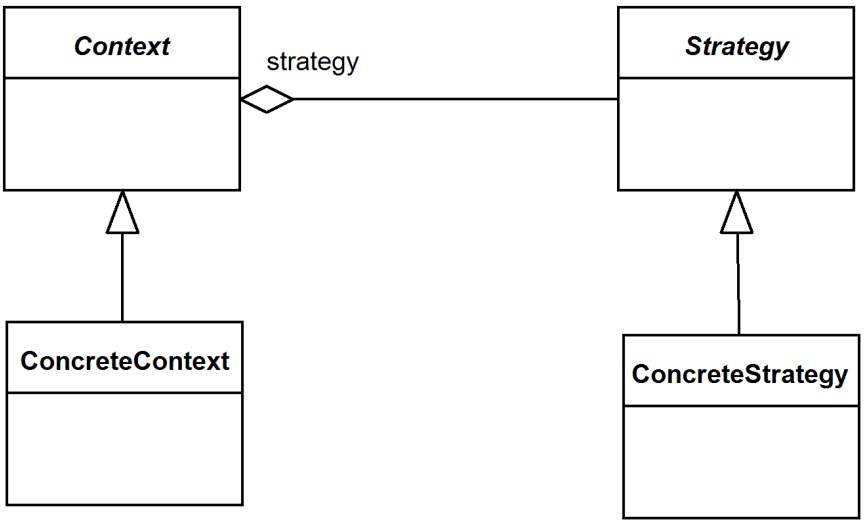
4. posluchač může být jeden, několik, nebo i celé skupiny
5. řešení pro problém, kdy máme jeden vstup, jehož data nebo akce ovlivňují další části programu a chceme, aby se tyto části upravovaly automaticky

5.4.4 Singleton (Jedináček)



Obrázek 5: Singleton (Jedináček)

- **Základní informace a vlastnosti:** Návrhový vzor Jedináček patří mezi Vytvářecí vzory, upravuje tedy způsob vytváření objektu. Tento vzor je řešením na problém kdy programátor chce v jeden okamžik mít vždy pouze jeden objekt dané třídy. Chceme například mít pouze jeden objekt nějakého důležitého registru, jeden objekt kontrolující nějaké externí zařízení. Ve všech takových případech by možnost vytvořit další instance této třídy mohla způsobit vážné problémy.
- **Definice:** Návrhový vzor Jedináček zajistí, že objekt dané třídy je vytvořen pouze jednou (resp. třída má vždy pouze jednu instanci) a zajišťuje přístup k tomuto vytvořenému objektu.
- **Použití:** Jedináček v podstatě nedefinuje žádné další třídy. Jeho jedinou starostí je úprava jedné už námi definované třídy a to tak, aby mohla být vytvořena pouze jednou. Jedináček popisuje tvar jedné metody třídy starající se o instanciování sebe sama (Jedináček tedy vytváří sám sebe). Další částí je pak proměnná, která udává, jestli už došlo k vytvoření aspoň jednoho objektu, zda-li tedy můžeme vytvořit další objekt nebo už ne. Samotná realizace se liší v závislosti na programovacím jazyce, neboť je tímto vzorem upravována samotná struktura třídy. Problémem při takovémto vytváření jsou vícevláknové aplikace – musíme tedy danou metodu synchronizovat.
- **Základní vlastnosti ve zkratce:**
  1. Vytvářecí vzor
  2. realizace závisí na programovacím jazyce
  3. řeší problém vytváření pouze jednoho objektu dané třídy
  4. vytváří sám sebe a udržuje si počet už vytvořených instancí
  5. musíme lehce upravit pro použití ve vícevláknových aplikacích



Obrázek 6: Strategy (Strategie)

5.4.5 Strategy (Strategie)

- **Základní informace a vlastnosti:** Návrhový vzor Strategie patří do kategorie vzorů Chování, tedy ovlivňující chování aplikace. Snažíme se o to, oddělit skupiny algoritmů (rodiny), které by se mohli v budoucnu měnit od sebe. Samotný klient je potom složen z těchto rodin, nepoužíváme dědičnost, ale skládání tříd dohromady. Klient sám určí, který algoritmus z této rodiny se použije – můžeme si dynamicky vybrat jakýkoliv algoritmus z této skupiny a nemusíme přitom nijak zasahovat do už napsaného programu (polymorfismus).

- **Definice:** Návrhový vzor Strategie definuje rodinu algoritmů, zapouzdří každého z nich a umožní, každý takový algoritmus vnitřně podle libosti měnit. Dovoluje tedy nezávislou změnu algoritmů na klientovi, který je využívá.

- **Použití:** Smyslem Strategy návrhového vzoru je zapouzdřit části programu, které se liší (rodiny algoritmů). Pokud tedy máme skupinu (rodinu) algoritmů, kde každý algoritmus řeší podobný problém, ale liší se realizací, pak je vhodné uvažovat o užití tohoto vzoru. Oproti jiným řešením nabízí velkou možnost změn i v budoucnu, bez toho aniž by jsme museli zasahovat do už vyzkoušeného a otestovaného kódu.
  Toho dosáhneme tak, že vytvoříme pro každou takovouto skupinu jedno rozhraní (abstraktní třídu, atd., záleží na programovacím jazyce), které použijeme k vytvoření potomků. Každý takovýto potomek se nazývá algoritmus, a celá skupina je pak rodinou algoritmů. Použití takové skupiny už je potom jednoduché, požadovanou třídu pouze z těchto rodin složíme a můžeme dynamicky měnit využívaný algoritmus. Tyto části jsou bezpečně odděleny (zapouzdřeny) od další části aplikace.

- **Základní vlastnosti ve zkratce:**

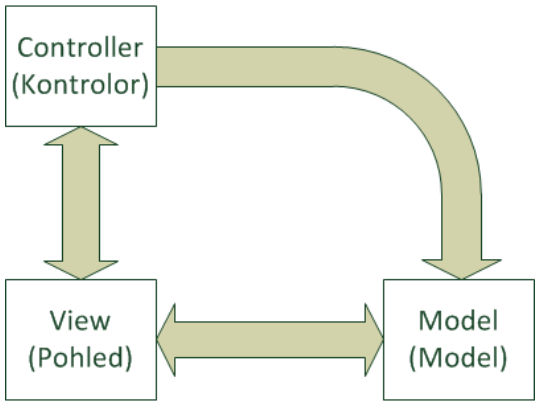
  1. vzor Chování
  2. slouží pro bezpečné, znovupoužitelné zapouzdření skupiny algoritmů
  3. klient je složen z těchto skupin
  4. v klientovi ovšem můžeme definovat i vlastnosti, stejné pro všechny potomky
  5. řešení na problém, kdy chceme, aby náš kód byl bezpečně rozšířitelný a upravitelný

6. svým chováním se částečně podobá návrhovému vzoru Stav

5.5 MVC (Model-Pohled-Kontrolor)

MVC se řadí mezi Složené vzory, avšak on sám je považován spíše za architekturu. MVC je dnes jeden z velmi používaných vzorů řešící, jak od sebe oddělit tři nezávislé úseky aplikace a v podstatě navrhnout základní kostru programu. Právě tuto architekturu využívá můj navržený systém.

- Model (Model)–udržuje data, stav a aplikační logiky programu
- View (Pohled)–obstarává část, kterou uživatel vidí
- Controller (Kontrolor)–stará se o komunikaci mezi Pohledem a Modelem, popřípadě obsahuje další výpočetní mechanismy aplikace



Obrázek 7: MVC–Model View Controller

Z Obrázku 7 jsou patrné vazby mezi jednotlivými bloky. Uživatel přistupuje k Pohledu. Pakliže uživatel provede nějakou změnu, je informován Kontrolor, který tuto situaci řeší, a pokud je třeba, kontaktuje Model. Pohled si od Modelu bere potřebná data a Model automaticky upozorňuje Pohled, pokud dojde ke změně v jeho stavu. Kontrolor může měnit zobrazení Pohledu.
Tuto architekturu můžeme realizovat pomocí tří návrhových vzorů.

1. Strom–realizuje část Pohledu (ta je složena z GUI komponent)
2. Pozorovatel–realizuje část Modelu
   - pokud Kontrolor změní jeho stav nebo data, je potřeba pomocí vzoru Pozorovatele informovat Pohled, popřípadě další části systému
3. Strategie–část Kontroloru
   - Kontrolor obsahuje strategie (jednotlivé rodiny algoritmů a algoritmy pro řešení dané situace
   - Pohled obsahuje referenci na tyto algoritmy a zobrazuje jejich výsledky–stará se tedy pouze o jejich prezentaci
   - Vzor Strategie tedy Pohledu poskytuje výsledky na uživatelův požadavek

Díky této architektuře dovedeme snadno **synchronizovat několik pohledů** zobrazující stejná data mezi sebou. Dále od sebe oddělíme různé funkční částí a tím zvýšíme znovupoužitelnost a budoucí snadnou rozšířitelnost aplikace.

6 Definování struktury problému a její optimalizace

Základním kamenem návrhu jakéhokoliv systému jsou samotná rozhodnutí, která určí jeho budoucí podobu. Tato rozhodnutí lze pochopit jako **problémy**, které je nutné řešit a které řeší i **návrhové vzory**.
Vybrání vhodného, případně žádného návrhového vzoru, potom často odpovídá porovnání našeho problému s definicí problému u návrhového vzoru.
Problémem zůstává textový popis návrhového vzoru (i když je tento popis standardizovaný, není zdaleka formalizovaný–tento nedostatek z částí odstraňuje popis v UML).
V mnoha případech je návrhový vzor velmi komplexní, a tak jeho použití závisí nejen na samotném problému, ale i dalších okolnostech jakými mohou být třeba negativní dopady použití vzoru nebo výhodnost spolupráce s dalšími vzory.
Dalším problémem je nutnost návrhový vzor, či jeho implementaci upravit před samotným použitím, a tím v mnoha případech zvýšit jeho efektivitu.
Taková rozhodnutí mohou být pro méně zkušeného návrháře nesnadná, a přitom v mnoha případech výrazně ovlivní kvalitu budoucího návrhu.
Mnou navržený systém by měl náročnost takového rozhodnutí snížit a poskytnout návrháři směr, jakým by měl daný problém řešit. Aby mohla být tato aplikace aspoň z části úspěšná, je nutné si uvědomit jakými metodami lze zkvalitnit návrh budoucího systému a tuto metodu se pokusit automatizovat.

6.1 Jazyk vzorů (Pattern language)

Jazyky vzorů **rozšiřují množství poskytovaných informací** o jednom či několika vzorech a tím výrazněji usnadňují identifikaci vhodnosti návrhového vzoru a způsob jeho implementace.
Zatímco popis vzoru (v našem případě návrhového vzoru) se soustředí pouze na tento vzor a jen omezeně popisuje jeho implementace a vhodnost ve větších strukturách, jazyk vzorů ukazuje jeden a v mnoha případech i několik vzorů právě v těchto větších projektech, a tím dává programátorovi často rozsáhlejší množství informací. V případě použití více návrhových vzorů, ukazuje tento jazyk jejich vzájemnou komunikaci.
Jazyků vzorů existuje celá řada, např. (P. Dyson, B. Anderson: State Patterns), který se zabývá GoF návrhovým vzorem Stav. Tento vzor je zde dekomponován, použit v mnoha případech i architekturách, s mnoha úpravami.
Použití takového jazyku je velmi výhodné, protože návrháři ukazuje situace, kdy byl daný vzor úspěšně použit.
Naše aplikace proto bude tento princip, i když zdaleka ne tak standardizovaně, používat a dovolí uživateli, aby si sám ukládal a navrhoval složitější softwarové struktury, kde daný vzor úspěšně použil, resp. daný problém nějakým postupem optimálněji vyřešil.[8]

6.2 Refaktorování (Refactoring)

V některých případech se ideálnější řešení problému objeví až ve chvíli, kdy je velká část aplikace už napsaná nebo navržená. Pro takové případy existují refaktorovací metody, které dovolují kód optimálně upravit, a tím v budoucnu zvýšit konkurenceschopnost (udržitelnost, rozšířitelnost,atd.) aplikace.[9]
Problémem takovéto metody je nutnost mít už alespoň částečně napsaný nebo navržený systém a z toho plynoucí neefektivnost takového vývoje.

Dnes existují aplikace, které dokáží pomocí těchto úprav zvýšit rychlost aplikace, či ji jiným způsobem optimalizovat. Optimalizace kódu z hlediska budoucí snadné rozšířitelnosti respektující principy objektově orientovaného programovaní je obecně úloha, která jde velmi těžko automatizovat.
Pro náš vytvářený systém se budeme chtít pohybovat na vyšší míře abstrakce, a tím získat větší přehled o funkčnosti aplikace a o daném problému bez toho, aniž bychom se zabývali zbytečnými drobnostmi.

6.3 Gramatika

Pod slovem gramatika si můžeme představit libovolný textový popis problému (například i přirozený jazyk je gramatika). Takovýto popis lze potom porovnat se známými řešeními za pomoci různých transformací a dospět k volbě vhodného řešení.
Problémem použití gramatiky je, že často uživatele nutí učit se poměrně komplexní jazyk. Jazyk pro popis problému nemůže být příliš jednoduchý, protože by nedovoloval popsat komplexnost problému, ovšem pokud bude příliš složitý, může být pro uživatele velmi těžké ho použít a pro návrháře aplikace vytvořit.
Takové řešení, pokud by bylo dobře navrženo, by bylo poměrně složité, na druhou stranu by se dalo dobře zautomatizovat a poskytovalo by také dostatečnou úroveň abstrakce.
Pro návrh systémů na dostatečné úrovni abstrakce se dnes převážně používají **vizuální jazyky**, např. UML. To je dáno standardizovaností těchto jazyků a hlavně srozumitelností takového popisu. V navrhovaném systému, který je zaměřen především na méně zkušeného uživatele, je toto řešení výhodnější.

6.4 Analýza požadavků (System requirements)

Další možností zvolení vhodného návrhového vzoru případně návrhu optimálnější softwarové struktury je co nejpodrobnější analýza požadavků. Analýza požadavků je dnes v podstatě důležitá pro všechny softwarové projekty a od její kvality se často odvíjí i kvalita výsledné aplikace.
Tuto metodu je velmi těžké automatizovat a výsledky této metody často také, protože v mnoha případech je analýza požadavků řešena textem.
Pokud je pro výsledky této analýzy použit diagram, například **Případ užití** jazyka UML, pak je možnost automatizace značně vylepšena. V tomto případě narážíme na problém, kdy součástí GoF popisu návrhového vzoru není tento diagram. U jiných vzorů nebo softwarových struktur, kdy je tento diagram použit, lze potom uživatelův diagram vhodně porovnávat s již dokončenými projekty, a tím usměrňovat a zkvalitňovat budoucí návrh aplikace. I zde je ovšem problém s množstvím informací v čistě **syrovém textu**, který značně zkreslí přesnost nalezených dat. Pro přesnější vyhledávání je proto nutno zvolit jiný diagram, který sám o sobě definuje lépe požadavky kladené od systému.

6.5 Vhodná metodika

Zvýšit kvalitu výsledného návrhu může zvolení správné metodiky. Tyto metodiky se ovšem velmi špatně automatizují, a tak v mé aplikaci nejsou použité. Na výstup mého systému na návrhářův požadavek lze ovšem tyto metodiky použít, a tím výrazně zvýšit kvalitu návrhu.
Metodika popisuje postup každého kroku vývoje softwaru tak, aby **výsledná aplikace odpovídala požadavkům** a její návrh byl co nejdůmyslnější. Těmito kroky může být analýza požadavků (popis jak správně takovou analýzu vést), návrh rozhraní a tříd, přechod mezi těmito kroky a mnoho dalšího.
Metodiky jsou sami o sobě velmi komplexní a volba metodiky zásadně ovlivňuje výsledný program.

6.5.1 UP (Unifed process)

Unified Process (UP) je metodologie od autorů UML. Samotné UML není spojené s žádnou metodologií, ale je schopné být využito všemi dnešními metodologiemi. UML je tedy obrazová část, kdežto UP je samotný proces při kterém požadavky na software přetvoříme v samotný program. I přes své blízké spojení s UML není UP na rozdíl od něj stále standardizován. Existuje i komerční verze UP, zvaná RUP od IBM, která převzala Rational Corporation v roce 2003.
Tato metoda využívá iterační proces návrhu. Vývoj samotného projektu je rozdělen do několika částí (analýza, design, test, atd.) a tyto části dále rozděluje na několik fází. Pro každý krok a fázi je důležitá jiná část projektu. Každá fáze je potom rozdělena na jednu nebo více iterací. Iterace obsahuje několik kroků (plánování, analýza a design, konstrukce, integrace a testy), jejímž výstupem je částečně kompletní verze systému. Každý krok iterace definuje, čemu by se měli tvůrci softwaru v dané chvíli věnovat (případně jak klást otázky). Myšlenka rozdělení vývoje vychází z principu, že je často mnohem jednodušší vytvořit menší funkční celky a ty potom spojit.
Použití návrhových vzorů a kvalita navržené softwarové struktury je velmi závislá na zkušenostech návrháře (týmu návrhářů). Použití návrhového vzoru je možné téměř v kterémkoliv kroku vývoje, ovšem samotné UP práci s návrhovými vzory nijak nedefinuje.[2]

6.5.2 POAD (Pattern oriented analysis and desing)

POAD je metodologie vytvořená autory Sherif M. Yacoubem a Hany H. Ammarem speciálně pro **vývoj softwaru pomocí návrhových vzorů**.
POAD využívá další vrstvu abstrakce při samotném návrhu a aplikaci vytváří už od počátků pomocí návrhových vzorů. Vnitřní struktura samotných vzorů není v počátcích vůbec podstatná. Návrhový vzor je zde chápan jako komponenta. Použití správných komponent je potom velmi závislé na dobře provedené analýze požadavků a dostatečné knihovně návrhových vzorů.
Tato metodologie definuje postup trasování vzorů až na úroveň tříd a popisuje vhodné úpravy na této úrovni tak, aby byly návrhové vzory optimálně spojeny.
Výhodou použití tohoto postupu je často kvalitnější OO kód (návrhové vzory už jsou často velmi dobře objektově navržené) a větší míra abstrakce celého návrhu. Nevýhodou je ale nutnost dobré znalosti návrhových vzorů.[7]

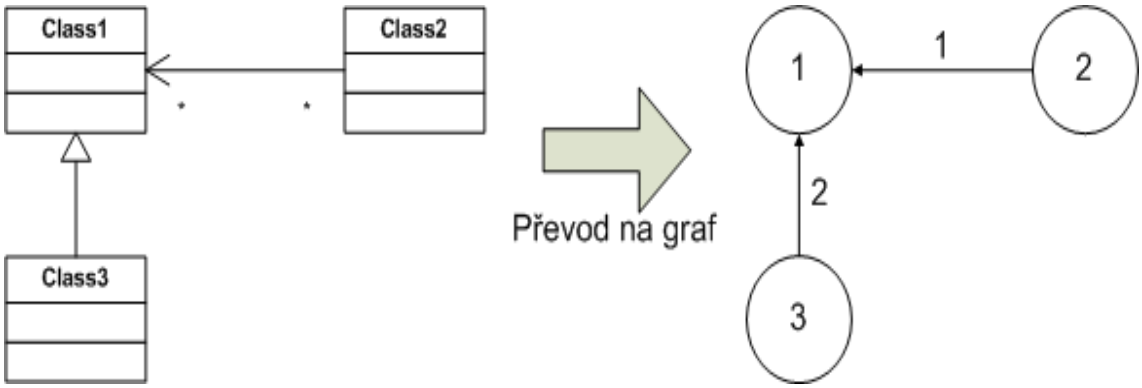
6.6 Použití UML diagramu tříd

Diagram tříd je jeden z nejpoužívanějších diagramů při návrhu OO systémů. Elementy a vazby použité v těchto diagramech také sami o sobě poskytují dostatek informací o navrhovaném systému. Pokud tedy vypustíme ostatní (textové, popisné, atd.) informace, stále získáme představu o řešeném problému.
GoF vzory a další OO návrhové vzory mají často ve standardu definován právě tento diagram, a tak není problém ho použít pro vybrání vhodného návrhového vzoru. Umožníme uživateli **nakreslit jeho problém** pomocí elementů tohoto diagramu, a potom tento diagram porovnáme s libovolným diagramem návrhového vzoru. Výsledek podobnosti nám určí vhodnost použití tohoto vzoru. Tento postup půjde poměrně dobře automatizovat a usnadní uživateli práci. Důležitým aspektem takového porovnávání je standardizovanost UML.
Aby jsme mohli tyto dvě struktury vůbec porovnávat musíme nejdříve dané diagramy převést na grafový problém a vybrat vhodný algoritmus pro samotné porovnávání.
Také musíme navrhnout strukturu knihovny diagramů. Ta bude obsahovat nejen diagramy návrhových vzorů, ale i další osvědčené postupy a jejich diagramy, které budou odpovídat počátečnímu problému, postupné úpravě tohoto problému a konečnému řešení.

I když diagram tříd sám o sobě poskytuje dostatek informací o systému, dojde přeci jenom při převodu na graf ke ztrátě mnoha údajů, které mohou zkreslit přesnost nalezeného výsledku. Další nevýhodou je poměrně velká časová náročnost takového porovnávání a nutnost vytvořit dostatečně obsáhlou knihovnu už vytvořených, optimálnějších řešení.

6.6.1 Převod UML na graf

Při převodu diagramu vzniká graf $G = (V, E, We, Wv)$ (viz Obrázek 8), kde vrchol V je chápán jako element třídy a hrany E potom reprezentují vazby mezi třídami. Graf G je díky možné orientaci vazeb mezi třídami také orientovaný. Ohodnocení hrany, We, symbolizuje typ původní vazby. Hodnoty ohodnocení nejsou v podstatě důležité, protože neupřednostňujeme některý typ vazby. Jde nám pouze o rozlišení jednotlivých druhů vazeb od sebe.
Pro výsledný graf G platí $E \subseteq V \times V$. I když element třídy podle UML povoluje mít vazbu sám na sebe, mi pro zjednodušení všechny smyčky a násobné hrany odstraníme (toto odstranění nám zrychlí algoritmus porovnávání grafů).



Obrázek 8: Převod UML diagramu na graf

Takový graf je potom v programu vyjádřen příslušným objektovým modelem na kterém probíhají potřebné operace a z kterého je konstruována potřebná matice. Řádek matice symbolizuje počáteční vrchol, sloupec potom cílový vrchol. Pokud $G[a][b] > 0$, kde a je příslušný řádek a b příslušný sloupec, pak mezi vrcholy existuje spojení a hodnota tohoto spojení určuje druh vazby mezi třídami. Jinak vrcholy nejsou spojeny $\Rightarrow$ ohodnocení vazeb je vždy kladné.
Matici předchozího grafu potom lze zapsat:

$$\begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 2 & 0 & 0 \end{pmatrix}$$

Nyní, když jsme diagram převedly na graf (matici), už můžeme v aplikaci vybrání vhodného návrhového vzoru plně automatizovat. Z čeho budeme vybírat nejoptimálnější řešení, bude záviset na velikosti a obsahu uložené knihovny správných postupů návrhů už vyřešených problémů a jednotlivých kroků těchto řešení.
V tomto bodě, když už víme, jakou metodu použijeme pro porovnávání dat, je důležité si uvědomit, co všechno budeme od výsledné aplikace požadovat.

7 Návrh aplikace

Na navrhovanou aplikaci budeme mít několik požadavků:

- musí dovolit ukládat už vyřešené postupy (postup může mít maximálně určený počet kroků)
- musí uživatelovi umožnit přehledně a snadno tyto postupy spravovat
- uživatel může zadat svůj problém pomocí elementů diagramu tříd (element třídy) a vazeb (asociace a dědičnost)
- tento problém dovolí porovnávat s uživatelovou knihovnou osvědčených řešení a zobrazí nalezené shody

Programovacím jazykem pro vytvoření aplikace bude **Java** z důvodu snadné přenositelnosti a použitého grafického rozhraní.

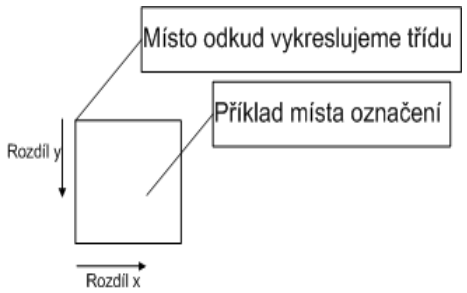
7.1 Panel kreslení

Pro vstupy i zobrazení dat z knihovny musíme vytvořit vlastní grafickou komponentu, kterou si pojmenujeme jako **panel kreslení**. Ta nám dovolí přehledně vykreslovat požadovaný prvek a reagovat na uživatelem spouštěné události.
Vykreslený prvek bude symbolizovat objekt daného prvku uložený v aplikaci. Každý prvek bude obsahovat i metodu pro své vlastní vykreslení. Volba mezi druhy vytvářených objektů (mezi třídami) bude umístěna ve vlastním panelu přehledně nad kreslícím plátnem. Aby bylo kreslení pro uživatele dostatečně příjemné, přidáme několik listenerů na určené události, které dovolí spravovat už nakreslené prvky přímo na plátně:

- úprava vlastností třídy (jméno, metody, atributy) nebo vazby (orientace) – dvojité poklepání
- změna velikosti třídy – klávesa R a levé tlačítko myši
- změna polohy třídy – levé tlačítko myši

Pro realizování všech těchto akcí, musíme vytvořit třídu, která bude udržovat podpůrné proměnné (samozřejmě mohou být tyto proměnné umístěny přímo ve třídě panelu, ale zbytečně jí budou znepřehledňovat). Především se jedná o:

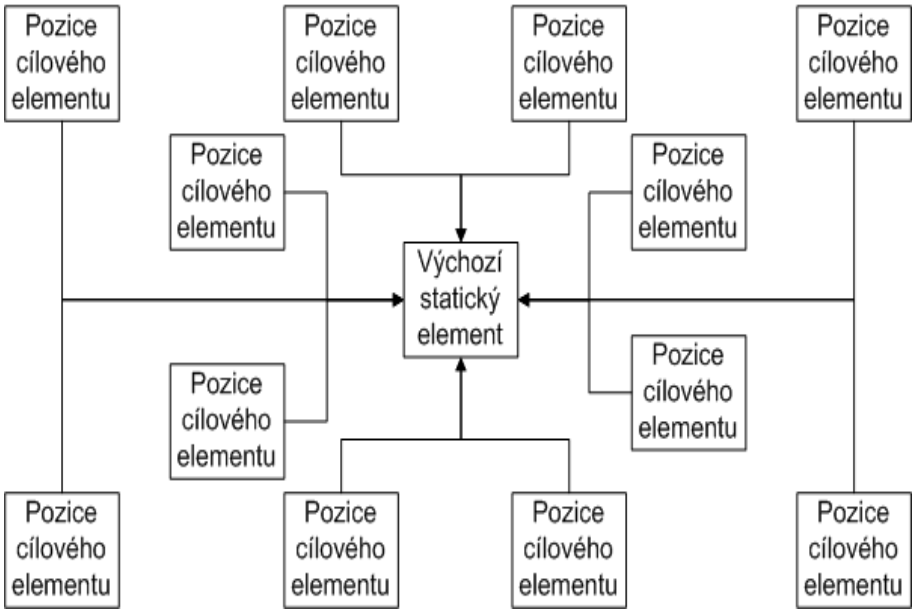
- první označená třída při vykreslování vazby (při označení druhé vykreslíme vazbu mezi nimi)
- třída je vykreslována podle pozice x a y v pravém horním rohu – aby uživatel mohl s vykreslením třídy manipulovat a nemusel vždy přesně označit myší roh třídy, je nutné spočítat rozdíl mezi označenou polohou a rohem (viz Obrázek 9) a tyto hodnoty uložit (hodnoty lze použít pro změny velikosti prvku i pro změny polohy)



Obrázek 9: Nutnost ukládat pomocné proměnné

7.1.1 Vykreslování vazby

Aby jsme dosáhli co největší přehlednosti mezi spojenými prvky, je vykreslování vazeb poměrně komplexní činností. Tvar vazby je dynamicky měněn a vykreslován podle pozice počátečního a cílového prvku (mění se nejen cílový a počáteční bod úsečky znázorňující vazbu, ale i místa lomení). Pokud je vazba orientována, pak je dynamicky překreslována i šipka. Zvolený systém vykreslování vazby je ukázán na Obrázku 10.



Obrázek 10: Vykreslování vazeb mezi statickými prvky

7.2 Objektový model vykreslovaných prvků

Už z předchozích úvah je patrné, že každý vykreslovaný prvek musí mít svou vlastní třídu, která definuje přehledně jeho vlastnosti. Veškeré prvky si můžeme rozdělit do dvou skupin:

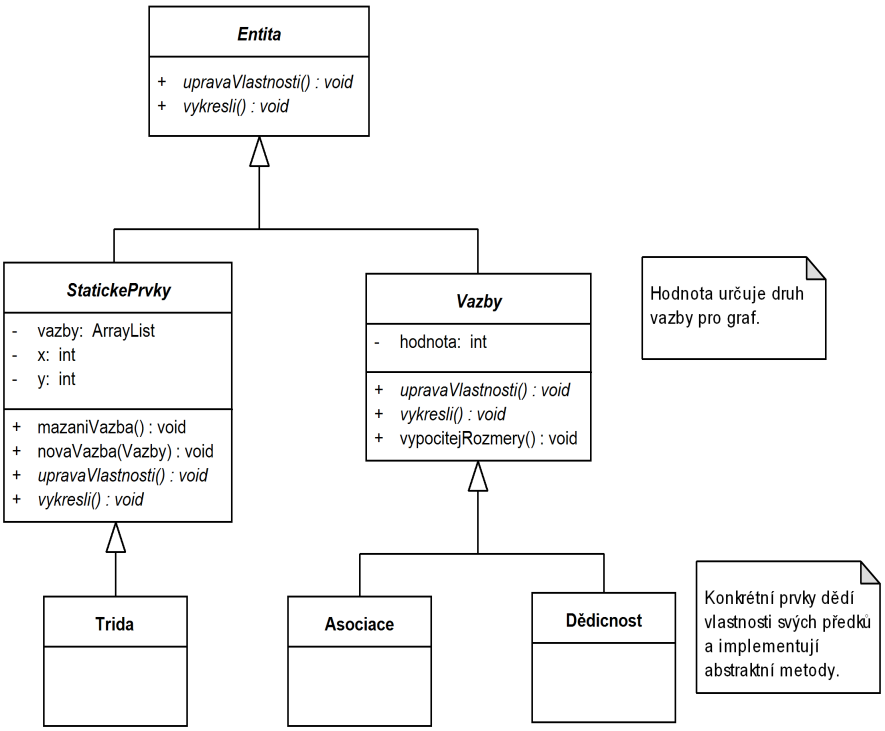
- statické prvky (například třída)
  - mohou na plátně existovat sami o sobě
  - jejich poloha je určena x a y souřadnicí
  - můžeme měnit jejich velikost i polohu
- vazby
  - vyjadřují spojení mezi dvěma statickými prvky
  - nemohou tedy na plátně existovat sami o sobě
  - jejich poloha je určena polohou spojovaných prvků

Abychom při práci s prvky dosáhli co největšího zapouzdření a nejvolnějšího spojení mezi třídami, vytvoříme ještě jednu třídu, kterou pojmenujeme Entita. Ta bude definovat společné vlastnosti statických prvků i vazeb a třídy Statické prvky a Vazby budou jejími potomky (viz Obrázek 11). Třídy konkrétních prvků jsou potom potomky těchto dvou tříd a musí implementovat některé jejich metody.
V celé aplikaci se pracuje s obecnou třídou Entity, výjimečně pak s třídami Statické prvky nebo Vazby. Tím jsou konkrétní prvky dostatečně zapouzdřeny (využití polymorfismu) ⇒ budoucí přidání dalších prvků si nevyžádá žádné změny v již napsaném zdrojovém kódu.

7.2.1 Převod na graf

Při převodu na graf je nutné z objektového modelu vytvořit matici. Pro kreslící panel musíme vytvořit třídu, která bude spravovat prvky v ní nakreslené. Každý kreslící panel bude potom inicializovat vlastní objekt této třídy a mít svou vlastní datovou vrstvu, kterou bude vykreslovat. Vzhledem k tomu, že metoda vykreslení každého prvku je už abstraktně vytvořena v Entitě (viz Obrázek 11), je datová vrstva naprosto oddělena od skutečného typu prvku.
Vytvoření matice potom probíhá podle následujícího postupu:

1: $matice[pocetStatickychPrvku][pocetStatickychPrvku] = 0$; {inicializace matice a nastavení počátečních hodnot na 0}
2: **for** i pro všechny $statickePrvky$ **do**
3: **for** j pro všechny vazby v i **do**
4: $matice[jOdkud][jKam] = jHodnota$; {vazba ví, odkud kam směřuje (objekty statických prvků), jako hodnota matice se použije atribut hodnota vazby}
5: **end for**
6: **end for**

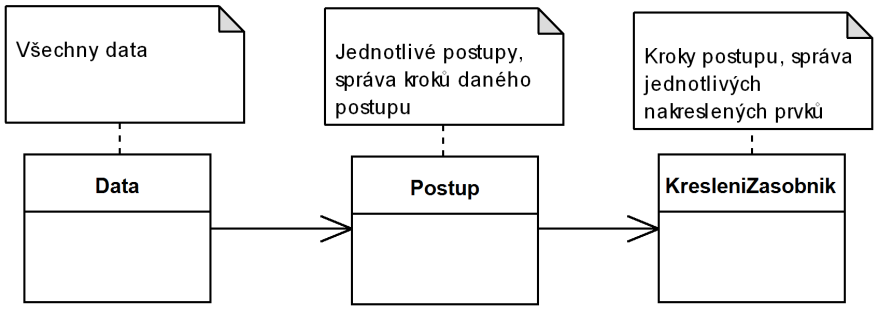


Obrázek 11: Objektový model prvků

7.3 Návrh knihovny postupů

Při návrhu datové vrstvy aplikace musíme myslet na její dostatečnou flexibilitu. Od aplikace požadujeme, aby uživatel mohl ukládat nejen nakreslené prvky jednoho panelu kreslení, ale aby těchto panelů mohl mít i několik najednou tak, aby se daly seskupovat do jednoho vyřešeného postupu (postup=problém), kde jednotlivý panel je krokem řešení (krok=postupné řešení problému).
Vytvoříme proto třídy, které budou takovouto strukturu reprezentovat (viz Obrázek 12). Třída Data bude obsahovat metody pro práci s jednotlivými postupy, bude také v budoucnu obstarávat ukládání a načítání dat z databáze. Třída Postup bude udržovat a spra-

vovat jednotlivé kreslící panely a třída KresleniZasobnik bude spravovat prvky kreslícího panelu.
Kreslící panel si nikam neukládá, co do něj bylo nakresleno. Pouze ví, který kreslící zásobník patří jemu, a z něj potom vykresluje prvky. Tato realizace je použita pro oddělení závislostí od pohledu a dat. Další výhodou tohoto rozdělení je budoucí ukládání dat (kdyby jsme nakreslené prvky ukládaly přímo do objektu plátna, museli bychom v budoucnu ukládat celé kreslící panely).



Obrázek 12: Správa dat

7.3.1 Správa knihovny postupů

Každá třída pro ukládání dat v aplikaci poskytuje i potřebné metody pro jejich správu. K atributům nikdy nepřistupujeme přímo.
Uživatel přistupuje k datům pomocí GUI (využití Java komponenty JTree, která dokáže přehledně vykreslit požadovanou strukturu dat–všechny postupy i jejich kroky). S postupy budeme chtít provádět několik operací:

- vytvoření nového postupu–samostatný formulář
- přejmenovávání postup nebo zásobníku (každý zásobník i postup má pro přehlednost své jméno)
- mazání zásobníků a postupů
- kopírování zásobníků mezi sebou–při kopírování budeme klonovat příslušné objekty (k tomu nám poslouží rozhraní cloneable a metoda clone() jazyka Java)
- přidávání dalších kroků (zásobníků) do už vytvořeného postupu

V pohledu (uživatelově GUI) může být více míst, kde budeme chtít zobrazit přehledy datové vrstvy. Narážíme proto na problém, jak je v pohledu všechny udržet aktuální (tak, aby obsah jednoho prvku odpovídal všem dalším). Pokud nastane situace, kdy uživatel prostřednictvím nějaké komponenty upraví data, potom musí i ostatní komponenty dostat upozornění, že došlo ke změně dat, a svá data upravit. Tohoto chování lze dosáhnout minimálně dvěma způsoby:

- vytvořením vlastního kontroloru pro tuto správu dat–příslušná GUI komponenta bude delegovat kontrolora a ten provede změnu nejen v datech, ale i všem dalším komponentám oznámí, že k takové změně došlo
- vytvořením synchronizační třídy–ta bude obsahovat příslušné synchronizační operace a všechny další GUI komponenty, které je nutné kontaktovat, pokud dojde ke změně v datech

Pokud se na oba postupy podíváme pozorněji, zjistíme, že jsou si velmi podobné. V obou případech využijeme základní princip návrhového vzoru Pozorovatel. V prvním případě bude objekt kontrolor spravovat všechny posluchače a provádět dané operace. V druhém případě si bude synchronizační objekt vytvářet přímo daná GUI komponenta. Ta si také sama provede všechny nutné operace a potom jen kontaktuje synchronizační objekt pro správu zbylých komponent. Toto řešení je použito v aplikaci, protože poskytuje větší volnost pro výběr posluchačů (každá komponenta může mít jiné posluchače). Tato výhoda je ovšem vykoupena horší přehledností než první řešení.
Toto řešení se týká pouze správy postupů a jejich jednotlivých kroků, u samotných prvků na kreslícím panelu bude využit princip MVC pro udržení synchronizace mezi panely, a tedy použití kontroloru.
MVC v tomto případě není použito, protože komponenta jazyka Java JTree, potřebuje složitější správu, než jakou můžou data pouhým oznámením o změně poskytnout (museli bychom tyto třídy více provázat). Proto vytváříme specializovaný objekt právě pro správu takovýchto přehledových stromů, který supluje funkci kontroloru modelu MVC pro každou danou komponentu.

7.4 Ukládání dat

Je nutné data ukládat proto, abychom k nim mohli zpětně přistupovat i po znovu spuštění aplikace a mimo jiné je i sdílet mezi několika uživateli (aplikacemi). Při využití jazyka Java se nám nabízí následující možnosti, jak data uložit:

- využijeme serializaci objektů – objekty můžeme ukládat do souborů a v případě potřeby je znova načítat, takový postup ovšem neposkytuje dostatečnou efektivnost pro správu dat
- využití objektově-relační databáze – nevýhodou je ovšem nutnost získávat z objektů potřebná data
  - pro ukládání a načítání dat z/do objektů si napíšeme vlastní datovou vrstvu (skupinu tříd), která nám zajistí abstrakci při práci s objekty a jejich daty, případně využijeme některý už vytvořený framework
  - dalším řešením je využití nástroje **Hibernate**
- použití objektové databáze – taková databáze by byla z hlediska aplikace nejvýhodnější a oproti relační databázi by nám ušetřila práci s relačním mapováním použitých objektů

7.4.1 Hibernate

Hibernate je nástroj, který dovoluje ukládat a načítat data v objektech jazyka Java do a z relační databáze za použití principu **ORM** (objektově relačního mapování). Tento nástroj velmi usnadňuje vývoj Java aplikací, kde má být jako persistentní úložiště dat použita relační (případně relačně-objektová) databáze.[21]
Pro použití tohoto nástroje je nejdříve nutné nastavit konfigurační soubor Hibernate. Potom už jen u ukládaných tříd nastavíme příslušné anotace (co budeme ukládat), případně provedeme toto nastavení v příslušném xml souboru a vytvoříme potřebné tabulky v databázi. Pomocí Hibernate dále uložíme/načteme potřebná data z/do databáze – pro práci s Hibernate většinou využijeme abstraktní vrstvu JPA (Java persistence API). Hibernate zde tedy obstarává funkci mapování mezi databází (tabulkami) a objekty.
Hibernate se často používá u webových aplikací na serveru, ale najde využití i u aplikací na desktopu. Pro svou funkčnost ovšem vyžaduje databázi, s kterou bude spolupracovat.

7.4.2 Databáze db4o

Databáze db4o je objektová databáze s licencí GNU GPL. Tato aplikace dovoluje naplno využít už vytvořenou objektovou strukturu aplikace. Není zde zapotřebí přidávání dalšího funkčního kódu do ukládaných tříd (ani anotací). Tato databáze je .jar Java balíček a po importování do naší aplikace můžeme jednoduše využívat její metody pro ukládání a načítání dat.[20]
Po kompilaci kódu tedy nepotřebuje žádnou další externí aplikaci jako suplikant databáze. Data jsou uložena ve formě souboru. Importované metody databáze potom dokáží z tohoto souboru efektivně získat a spravovat dané objekty, případně je do něj ukládat, mazat, aktualizovat, atd.
Po otevření spojení (databáze otevře soubor) dojde k zamknutí otevíraného souboru. Databáze si pamatuje referenci všech načtených objektů až do uzavření spojení a při ukládání dat kontroluje objekty, které je nutné pouze aktualizovat (nedojde tedy k redundantnosti dat).
Nevýhodou této databáze je, že při podstatné změně tříd v aplikaci, může dojít k nekonzistenci s uloženými objekty a ke ztrátě všech uložených dat. Další nevýhodou je silné provázání s použitým programovacím jazykem.
Příklad obsluhy databáze db4o:

```
1  //potřebné importy
2  import com.db4o.Db4oEmbedded;
3  import com.db4o.ObjectContainer;
4  import com.db4o.ObjectSet;
5  import com.db4o.config.EmbeddedConfiguration;
6
7  //vytvoření nové konfigurace db
8  EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();
9  /* při načtení objektu se načtou i všechny objekty v něm uložené, atd. až do dané hloubky */
10 config.common().activationDepth(10);
11 /* nastavení kaskádních úprav daných tříd */
12 config.common().objectClass(Postup.class).cascadeOnUpdate(true);
13 config.common().objectClass(KresleniZasobnik.class).cascadeOnUpdate(true);
14 config.common().objectClass(Entita.class).cascadeOnUpdate(true);
15 //načtení souboru s ˜příslušnou konfigurací
16 db = Db4oEmbedded.openFile(config, soubor.getPath());
17
18 ////////// ukládání dat //////////
19 try {
20   db.store(ukladanyObjekt); /* nejjednodušší možnost ukládání dat do databáze
21   (souboru)->databázi prostě poskytneme příslušný objekt */
22 } catch (Exception e) { // při ukládání může nastat chyba
23    // reakce na chybu
24 }
25 /////// načítání dat //////////
26 try {
27    ObjectSet vysledky = db.queryByExample(new Postup()); /* nejjednodušší
28    načítání objektů -> podle požadovaného objektu (metoda vrací list objektů) */
29    for (Object o : vysledky) { //nacteme data
30         // o obsahuje požadovaný objekt;
31     }
32 } catch (Exception e) {
33         // reakce na chybu
34 }
35 /////// ukončení spojení //////////
36 db.close();
```

Uvedený příklad načítání dat využívá nejjednodušší typ dotazu, dotaz pomocí příkladu (query by example). Databáze db4o dovoluje využít ještě další dva druhy dotazů. Prvním z nich je přirozený dotaz (native query), který je preferován v případě, že dotaz pomocí příkladu je nedostatečný.

```
List<Postup> vysledek = db.query(new Predicate<Postup>() {
        public boolean match(Postup postup) {
                // vložení podmínky
                return true
        }
});
```

Poslední možností dotazu je pomocí SODA API.

```
Query query−db.query(); //dotaz pomocí SODA API
query.constrain(Postup.class); // omezení
query.descend("atribut").constrain(100); //omezení na atribut třídy
ObjectSet vysledek−query.execute(); //vykonání dotazu a˜získání dat ve výsledku
```

Protože výsledná aplikace potřebuje mít všechny data přístupná hned po načtení (to je nutné vzhledem k vytvoření přehledu všech postupů pro uživatele i pro samotné vyhledávání), stačí nám použití dotazu pomocí příkladu. Všechny postupy z databáze získáme, pokud jako příklad použijeme objekt Postupu se všemi atributy o hodnotě null.
Z předchozí kapitoly víme, jak bude vypadat objektový model dat. Třída Data bude obsahovat metody pro načtení, uložení a vytvoření spojení s databází. Abychom zajistili dostatečnou abstrakci ovládání těchto metod od uživatele, vytvoříme ještě další třídu OvladaniDB. Ta bude zpracovávat komplexnější činnosti s databází a bude si udržovat informaci o tom, jaký soubor máme právě otevřen. Protože chceme uživateli dovolit otevřít v jednu chvíli pouze jedno spojení, bude tato třída využívat návrhový vzor Jedináček.

7.5 MVC a použití návrhových vzorů

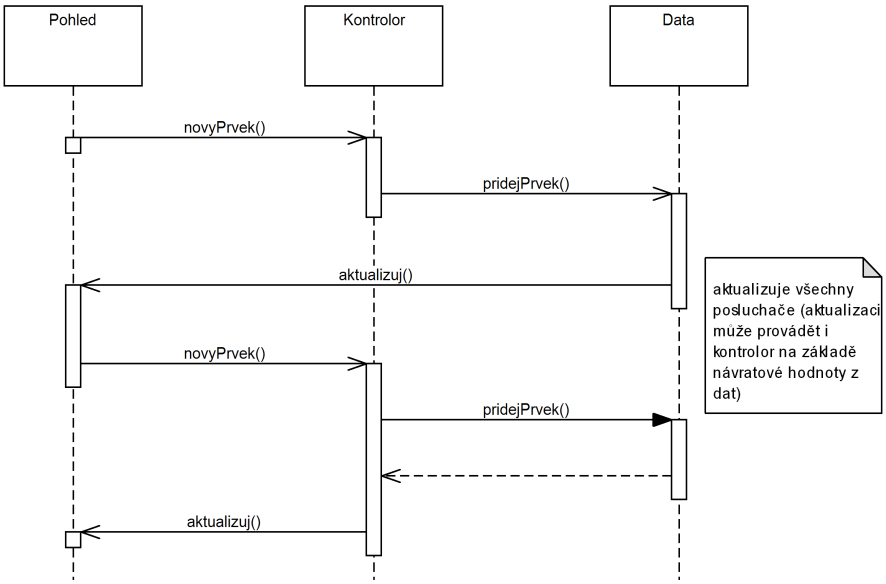
Už z předchozích rozhodnutí je možná patrné, že aplikace bude využívat architekturu MVC. O jejích výhodách jsem se zmínil již na začátku práce a zde bych rád ukázal jejich realizaci.
Aplikaci rozdělíme na tři funkční bloky:

- Pohled – vše co uživatel vidí. Pro většinu složitějších komponent vytváříme vlastní třídu (tím sice velmi zvýšíme počet tříd dané aplikace, ale také zvýšíme přehlednost a rozšířitelnost programu). Složitější komponenty obsahují část funkcionality kontroloru (tím podstatně snížíme velikost kontroloru a umožníme provádět některé operace už na úrovni Pohledu)

- Kontrolor – obsahuje podstatnou část veškeré funkcionality systému. Pohled pouze deleguje kontrolor o provádění daných operací. Zde přichází ke slovu použití různých strategií a návrhový vzor Strategie.

- Model – datová vrstva systému. Ta už zde byla poměrně rozebrána. Vzhledem k MVC je nutné zdůraznit, že to jsou právě data, která upozorňují Pohled na změnu (toto upozornění může v některých případech provádět i kontrolor, jak je patrné z Obrázku 13) a tím zajišťují aktuálnost Pohledu a možnost více-oknové aplikace.

I když použití MVC je v mnoha směrech výhodné, může mít i některá negativa. Je nutné aplikaci velmi dobře otestovat a funkčně dobře propojit tyto tři bloky (realizovat mezi bloky co nejvolnější vazby, a tím snížit budoucí závislosti–toho lze částečně dosáhnout užitím rozhraních, případně abstraktních tříd). V mnoha případech MVC způsobí „nakynutí“

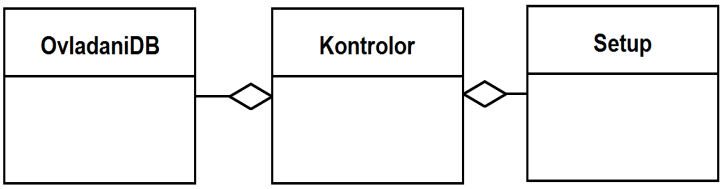


Obrázek 13: Sekvenční diagram MVC s rozdílnými přístupy aktualizace

celého programu a pro mnoho projektů, kde se vůbec nepočítá s jejím použitím jako vícepohledové, je jeho nasazení často zbytečné. V mé aplikaci je použití MVC spíše studijního rázu, než že by ho charakteristika problému vyžadovala.

7.5.1 Kontrolor

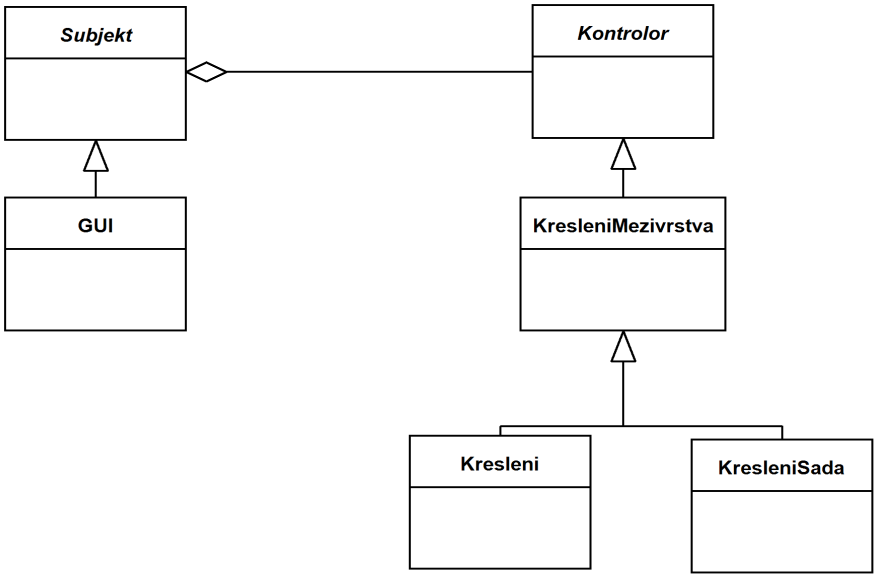
Kontrolor symbolizuje hlavní funkční jednotku aplikace. Nejenže má obstarávat většinu operací, ale stará se i o synchronizaci pohledů.
Aby nebyl kontrolor příliš rozsáhlý a dodržoval jeden z principů oo programování (každá třída by měla dělat jen jednu věc), je využita možnost skládání tříd (viz Obrázek 14). Objekt kontrolor tedy v některých případech **deleguje** jiné objekty, aby za něj provedli příslušné operace (těmi mohou být například ovládání databáze a nebo načtení počátečních setup dat). Poskytuje také přístup k těmto objektům přes své set/get metody.



Obrázek 14: Skládání kontroloru

Pro dodržení počátečního rozdělení aplikace na funkční bloky, je nutné, aby část Pohledu, případně dat, obsahovala referenci (využívala) pouze objekt kontrolor (nebo jeho potomky). Přes delegace, dědičnost a skládání tříd potom můžeme realizovat libovolnou operaci napříč celým Kontrolor blokem.
Pro oddělení použitých algoritmů je použit **návrhový vzor Strategie**, který dovoluje zapouzdřit a dynamicky měnit různé strategie použité Pohledem (viz Obrázek 15).
Objekt GUI přesně definuje skladbu okna a co uživatel vidí. Třída Subjekt potom slouží jako předek všem pohledům a je složena i z reference na objekt kontrolor, který symbolizuje abstraktní třídu všech strategií. Třída KresleniMezivrstva je vložena mezi strategie

a kontrolor z důvodu, abychom přesunuly společné kreslicí metody do jedné třídy, využili dědičnost a zpřehlednili jsme kontrolor.



Obrázek 15: Kontrolor–vzor Strategie

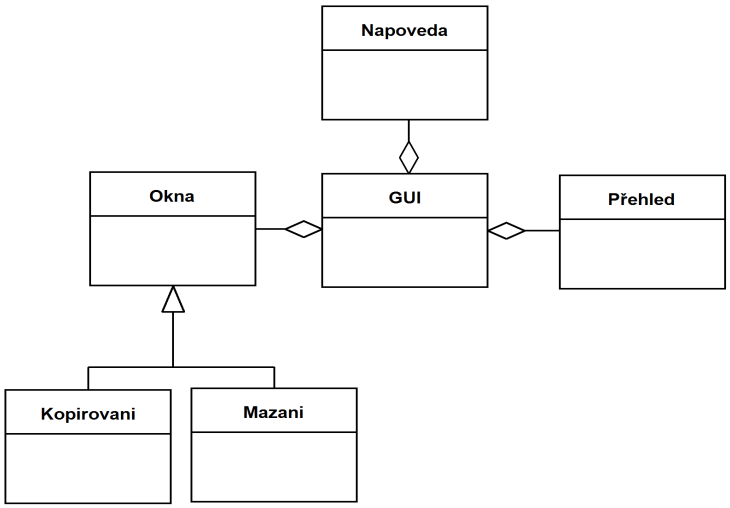
Různé strategie jsou v aplikaci chápány jako rozdílné způsoby práce s kreslícími panely při kreslení. Oblast této správy je pro aplikaci naprosto základní a v budoucnu je dosti možné, že právě v těchto algoritmech může dojít ke změně nebo k přidání dalších druhů kreslení. Díky použití tohoto vzoru, který využívá polymorfismus, budou nutné úpravy v již vytvořeném a otestovaném kódu minimální.
Rozdílné strategie zde mohou být potřeba například z těchto důvodů:

- rozdílný přistup k datům a prvkům v zásobníku – tento důvod je použit pro rozdělení kreslení a kreslení sady v mém systému. Zatímco kreslení je osamocený panel, sloužící pro vyhledávání a jeho pozice v datech je předem známá, kreslení sada pracuje často s celou sadou kreslících panelů
- uživatele chceme upozornit před smazáním nebo přidáním prvku
- správu můžeme realizovat dalšími jinými způsoby, můžeme omezit přidávání prvků na panel a upravit další operace, které jsou definovaný ve společném rozhraní kontroloru

Přepínání mezi strategiemi může potom být realizovaná podle uživatelova požadavku (jakou strategii bude chtít použít) a nebo použít automatické měnění strategií. Chce-li například uživatel vyhledávat, potom přepneme kontrolor jenom na kreslení a dovolíme mu spravovat data ve vyhledávacím kreslicím panelu. Pokud pracujeme nad uloženými postupy, přepneme kontrolor na kreslení v sadě.

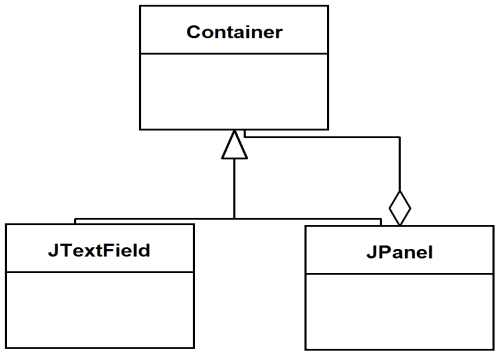
7.5.2 Pohled

Pohled symbolizuje všechny komponenty zobrazované uživateli. Třída GUI popisuje jakým způsobem budou dané komponenty poskládány a zobrazeny.
Pro většinu složitějších komponent je vytvořena vlastní třída. Pohled je složen z objektů těchto tříd (viz Obrázek 16). Vzhledem k použití MVC můžeme mít s minimálními úpravami více různě poskládaných pohledů.



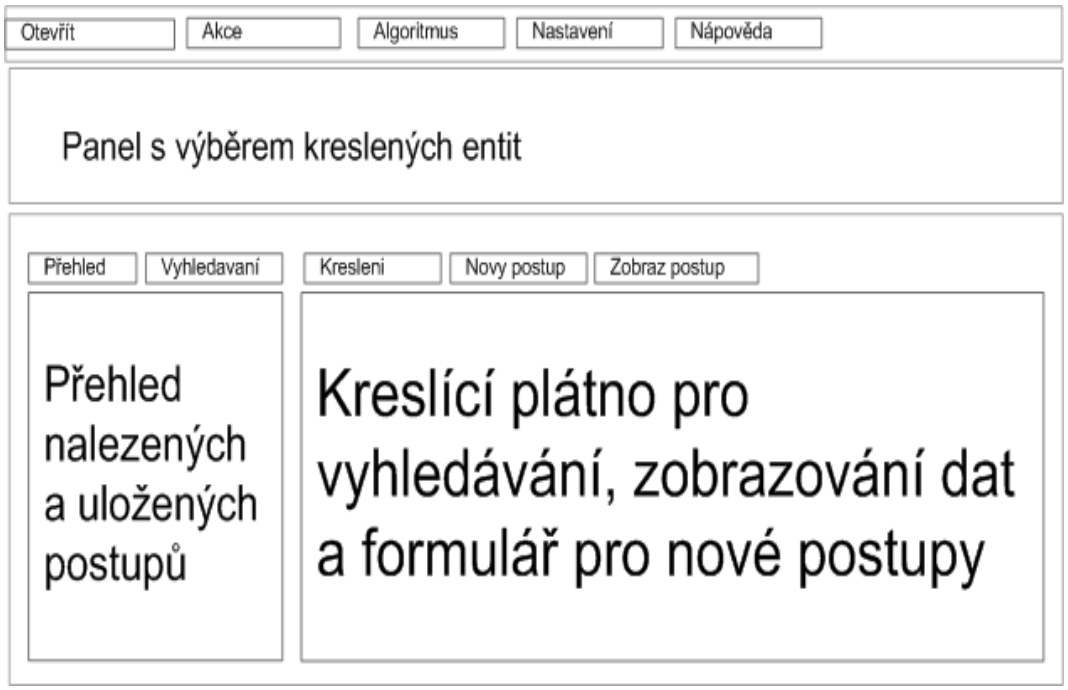
Obrázek 16: Složení pohledu

Skládání jednotlivých komponent do sebe je realizováno pomocí **návrhového vzoru Strom** (Composite). Ten je obsažen už přímo v programovém prostředí Javy, a není proto nutné ho vytvářet znovu. Se znalostí základního principu tohoto vzoru si dovedeme udělat velmi jasnou představu, jak je toto skládání v Javě realizováno (viz Obrázek 17, kde je velmi zjednodušená ukázka realizace tohoto vzoru v Javě).



Obrázek 17: Příklad skládání komponent v Javě

Pro návrh grafického uživatelského rozhraní lze použít **drátový model** (viz Obrázek 18), který dovolí v této fázi ukázat, jak bude toto rozhraní vypadat, a zda-li se v něm bude uživatel orientovat. Tuto část uživatel přímo uvidí a bude to v podstatě jediná část celého modelu, s kterou bude komunikovat. Díky použití modelu MVC můžeme mít takovýchto rozhraní hned několik, synchronizovaných mezi sebou navzájem.
Předem si připravíme rozvržení menu, jestli budeme uživateli umožňovat aplikaci nějak dodatečně nastavovat, nebo umožníme měnit použitý vyhledávací algoritmus. Podstatnou otázkou je, zda-li bude aplikace obsahovat nápovědu. Položku menu Otevřít potom navážeme na obsluhu databáze.
Z drátového modelu také vidíme, že nalezené výsledky budeme zobrazovat ve stejném panelu jako obsah databáze. Mezi těmito položkami se tedy budeme přepínat. To samé platí pro kreslící plátno určené pro vyhledávání postupů a pro plátna, na které budeme už uložené postupy zobrazovat.
Panel, z kterého budeme vybírat aktuálně kreslenou entitu, přehledně umístíme nad plátna a pod menu aplikace. Uživatel tedy nebude muset nějak zdlouhavě hledat kreslené prvky



Obrázek 18: Drátový model grafického rozhraní

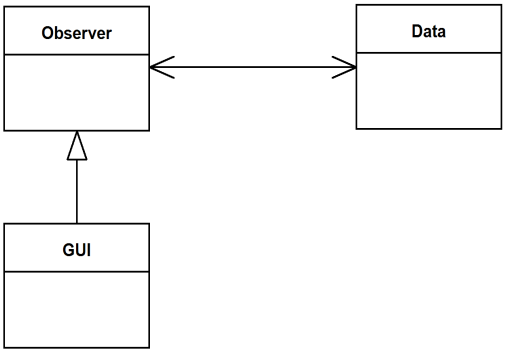
a složitě se k výběru proklikávat.
Možnosti nastavení a ukázky nápovědy už přímo souvisí s realizací aplikace, a proto se jim budu věnovat až později.

7.5.3 Model

U datové vrstvy MVC je nutné zmínit spojení mezi daty a pohledem. Ze strany pohledu je toto spojení zřejmé, pohled získává data z modelu. Jak ovšem upozornit pohled nebo několik pohledů, že došlo ke změně dat. Pokud se zaměříme na vykreslená data na jednotlivých panelech, mohli bychom samozřejmě počkat na další překreslení. Toto řešení není ale často dostatečně flexibilní.
Pro řešení tohoto problému využijeme **návrhový vzor Posluchač** (viz Obrázek 19). Data budou vědět, komu všemu poskytují informace, a při jejich změně upozorní i všechny registrované posluchače. Pro toto spojení je charakteristické, že je velmi volné (data o posluchačích ví jen opravdu minimální množství informací). Po realizaci tohoto spojení je už velmi jednoduché upozorňovat pohled nejen na změny, ale také na chyby při čtení dat, otevírání databáze, atd..
Pro všechny data v tomto případě není potřeba vytvářet společné rozhraní, čímž se diagram tříd mírně rozchází s uvedeným diagramem na začátku práce. Definice problému a hlavně použité řešení ovšem odpovídá danému návrhovému vzoru, a proto o tomto řešení můžeme hovořit jako o použití vzoru Pozorovatel.

8 Vyhledávání a porovnávání diagramů tříd

Pro vyhledávání bude použit uživatelův vstup (diagram tříd UML). Tento diagram bude převeden na graf a následně porovnán s již uloženými správnými postupy problémů a jejich optimálních řešení. Na základě shody bude vybrán nejoptimálnější postup a daný krok postupu. Tím uživatel získá přehled o tom, jak by měl pokračovat podle už osvědčeného postupu.



Obrázek 19: Realizace vzoru Posluchač

Pokud nalezneme shodu, ale navrhované řešení nebude odpovídat požadovanému, je tu možnost, že uživatel svůj problém nakreslil jinak než v minulosti a nebo v databázi, řešení tohoto problému není (lepší a odpovídající graf).
Vyhledávání má snížit počet možných budoucích řešení a napovědět uživateli, jak v minulosti řešil podobný problém. Nalezené řešení proto nemusí vždy odpovídat aktuálnímu požadovanému řešení. Tato nepřesnost ve vyhledávání je dána nejen nedostatkem přenášených informací v grafu, ale i automatizací celé úlohy. Proto například diagram tříd pro návrhový vzor Pozorovatel bez jednoho rozhraní a jiným typem vazby bude aplikací chápan jako jiný problém, než vyjadřuje standardizovaný diagram tohoto vzoru a to i přesto, že oba řeší stejný problém trochu rozdílným způsobem. Zde je vidět důležitost dostatečně rozsáhlé knihovny a možnost budoucího rozšíření aplikace o další diagramy, které by zvýšily množství přenášených informací.
Pro výsledný systém je nutné myslet na budoucí snadné rozšíření o další použité algoritmy. Toho lze dosáhnout společným rozhraním pro všechny vyhledávací algoritmy (využití polymorfismu). Algoritmy budou umístěny v zásobníku a výběr nebo přidání dalšího algoritmu bude otázka několika řádků nového kódu. Také je důležité nalezené shody přehledně zobrazit a vyznačit nejoptimálnější řešení.

8.1 Pojmy z teorie grafů

Problém porovnávání řešíme jako grafovou úlohu a je proto důležité vysvětlit ze začátku několik grafových pojmů, které budou použity pro vysvětlení principů algoritmů. Pro složitější pojmy je součástí vysvětlení i příklad, který v mnoha případech souvisí s výsledným řešením problému a můžeme se tak na něj v dalším popisu odkázat.

8.1.1 Graf

Definice i znázornění grafu jak vizuálně, tak v paměti počítače například maticově už bylo částečně zmíněno a zde tedy popíšeme zbytek potřebných informací.

- $G = (V, E)$–vyjádření grafu (graf je tvořen vrcholy a hranami)
- $E \subseteq \binom{V}{2}$–hrana je tvořena dvouprvkovou neuspořádanou množinou vrcholů (platí pro neorientované grafy)
- $E \subseteq V \times V$–hrany jsou uspořádané dvojice vrcholů (platí pro orientované grafy)
- předchozí dvě tvrzení neplatí pro grafy s vícenásobnými hranami a smyčkami–takový graf označujeme jako multigraf nebo pseudograf

Vizuální znázornění grafu jsme mohli vidět už na předchozím Obrázku 8 a graf bude znázorněn také na dalších popisných obrázcích.
Graf v počítači můžeme vyjádřit mnoha způsoby, a abych na následujících příkladech demonstroval jejich rozdílnost, je použit pro všechny případy stejný graf.

- uložení množin vrcholů V a hran E
  - $V = \{1, 2, 3, 4\}$ a $E = \{\{1, 2\}, \{2, 3\}, \{3, 4\}, \{4, 1\}, \{1, 3\}\}$
  - hrany E jsou vyjádřeny neuspořádanou množinou ⇒ neorientovaný graf
  - můžeme jednoduše vyjádřit násobné hrany a smyčky
  - nedovoluje vyjádřit ohodnocení hrany
- matice sousednosti
  - $$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$
  - dovoluje zobrazit smyčky, případně stupeň vrcholu
  - nedovoluje v této podobě vyjádřit násobné hrany
  - vhodná i pro orientovaný graf–neorientovaný graf bude mít tuto matici symetrickou
- incidenční matice
  - $$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$
  - řádek matice symbolizuje hranu (hrany je proto nutné očíslovat, případně jim přiřadit nějaké pořadí)
  - dovoluje vyjádřit smyčky i násobné hrany
  - dovoluje vyjádřit i stupeň vrcholu
  - použitelná pro orientovaný i neorientovaný graf

Protože v aplikaci budeme potřebovat ukládat ohodnocení hrany a už víme, že budeme ignorovat (respektive ani nedovolíme nakreslit) vícenásobné hrany a smyčky, bude nám pro práci s grafem nejlépe vyhovovat matice sousednosti.

8.1.2 Úplný graf

Úplný graf je takový, pro který platí $N = 2^{\binom{v}{2}}$, kde N je celkový počet vazeb a v je počet vrcholů grafu. Vizuálně úplný graf poznáme tak, že každý vrchol je spojen hranou s každým jiným vrcholem (tedy kromě sebe samého).

8.1.3 Podgraf

Podgraf $G_1 = (V_1, E_1)$ grafu $G = (V, E)$ definujeme jako $G_1 \subseteq G$. Podgraf je tedy **podmnožinou původního grafu**. Speciálním případem podgrafu je **indukovaný podgraf**, pro který platí $E_1 = E \cap \binom{V_1}{2}$, obsahuje tedy všechny hrany na podmnožině vrcholů podgrafu původního grafu.

8.1.4 Cesta a souvislost grafu

Cesta je množina, resp. uspořádaná posloupnost P, vrcholů V, ve které se mohou opakovat hrany. Definice cesty je důležitá pro pochopení souvislosti grafu.
O grafu můžeme prohlásit, že je souvislý, pokud existuje cesta pro každé dva vrcholy x a y. Každý maximální souvislý podgraf grafu G se nazývá **komponenta grafu**. Komponenta grafu se dá vyjádřit množinou vrcholů, které jsou spolu v relaci, resp. ekvivalenci (relace splňuje pravidla na symetrii–$x \sim y$, reflexi–$x \sim y \Rightarrow y \sim x$ a tranzitivitu–$x \sim y \wedge y \sim z \Rightarrow x \sim z$).
Už z popisu ekvivalence vyplývá, že zatímco u neorientovaného grafu musí platit vždy, u orientovaného platit nemusí (cesta z x do y existuje, ale cesta z y do x už existovat nemusí). Pro orientované grafy tedy zavádíme termín **silně souvislá komponenta**, která v podstatě odpovídá komponentě u neorientovaného grafu, tudíž, pokud existuje cesta z x do y, u silně souvislé komponenty existuje i cesta z y do x, kde x a y jsou libovolné dva vrcholy silně souvislé komponenty.

8.1.5 Strom

Za strom se dá označit každý souvislý graf neobsahující kružnici (pokud z libovolného vrcholu x do y existuje více jak jedna cesta $\Rightarrow$ strom splňuje podmínku jednoznačnosti cesty). O stromu se dá dále prohlásit, že se jedná o maximální graf bez kružnice a zároveň minimální souvislý graf (přidáme hranu – vznikne kružnice, odebereme hranu – graf už nebude souvislý). Pro strom také platí Eulerův vzorec $\mid V \mid = \mid E \mid +1$.
Kostra grafu je jeho stromem, pokud výsledný strom má stejný počet vrcholů jako původní graf.

8.1.6 Složitost algoritmů

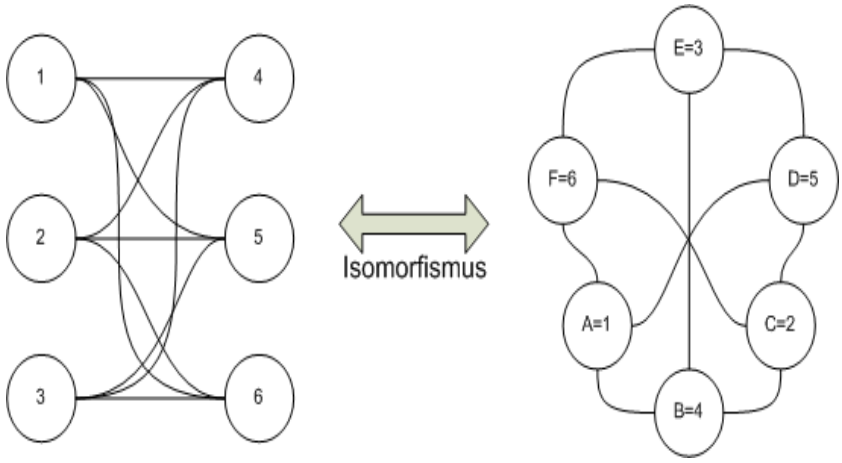
Pro práci s algoritmy, nejen grafovými, je vždy podstatná jejich složitost. Ta vyjadřuje časovou náročnost algoritmu a je přímo úměrná době nutné k získání požadovaného výsledku. Každý algoritmus by měl splňovat kritérium na svou konečnost, pokud je algoritmus příliš složitý, pak je vhodné omezit vstupní data či výpočet provést paralelně.
Pro popis asymptotické složitosti použijeme O notaci $\Rightarrow T(n) = O(f(n))$ a tedy čas běhu algoritmu odpovídá řádu růstu funkce f(n), která charakterizuje počet elementárních operací nad vstupními daty o počtu n.
Pro snazší pochopení zápisu složitosti uvedu pár příkladů:

- O(1) – elementární operace
- O($log_2 n$) – vyhledávání prvku v seřazeném poli pomocí metody půlení intervalu
- O(n) – vyhledávání prvku v neseřazeném poli lineárním vyhledáváním (lineární složitost)
- O(n^2) – procházení matice (kvadratická složitost)

Pro algoritmus jsou nejméně vhodné složitosti exponenciální – $O(e^n)$ a faktoriální – $O(n!)$. Pro rostoucí n je při těchto složitostech obrovský růst T(n) – doby nutné k výslednému výpočtu. Problémy, které nelze řešit a ověřit jinak, než pomocí algoritmů s větší než polynomiální složitostí, označujeme jako NP-úplné problémy. Další třídy problémů jsou P–řešitelné v polynomiálním čase a NP–neřešitelné, ale verifikovatelné v polynomiálním čase. Každý NP problém je redukovatelný na NP-úplný problém.

8.1.7 Isomorfismus

Isomorfní grafy $G = (V, E)$ a $G_1 = (V_1, E_1)$ jsou takové, pro které platí $\{x, y\} \in E$, právě když $\{f(x), f(y)\} \in E_1$ a zobrazení $f : V \to V_1$. Jinak řečeno, z původního grafu G můžeme získat n isomorfních grafů G_1 a to přejmenováním vrcholů V na V_1 tak, aby odpovídalo zobrazení jednotlivých hran E do E_1 (viz Obrázek 20). Pokud jsou dva grafy isomorfní, můžeme psát, že $G \cong G_1$. Z popisu isomorfismu vyplývá jedna z jeho vlastností a to, že dva isomorfní grafy mají stejnou posloupnost stupňů vrcholů (stupeň vrcholu vyjadřuje počet vycházejících hran z vrcholu, u orientovaných grafů potom rozlišujeme výchozí a příchozí hrany). Rozhodnutí, zda-li jsou dva grafy isomorfní, je obecně těžká úloha.
Zatímco zjistit jestli jsou dva grafy isomorfní je těžká úloha, rozhodnutí, zda-li dva grafy nejsou isomorfní, je často mnohem jednodušší problém. Zde můžeme využít vlastností isomorfismu pro určení, že dva grafy nejsou isomorfní. Touto vlastností je už zmíněná posloupnost stupňů vrcholů a to její velikost, pořadí a jednotlivé stupně vrcholů (stačí, aby dva grafy byly rozdílné pouze v jedné z těchto vlastností, a o grafech můžeme prohlásit, že nejsou isomorfní).



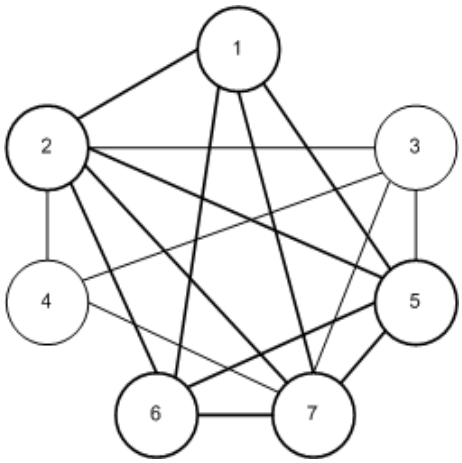
Obrázek 20: Isomorfismus

8.1.8 Klika grafu

Klika neorientovaného grafu G=(V,E) je množina vrcholů $C \subseteq V$ takové, že každý podgraf grafu G indukovaný vrcholy C je kompletní, resp. každá klika **neorientovaného** grafu G je jeho **úplným** podgrafem (viz Obrázek 21).
Při určování klik rozlišujeme její dva typy:

- největší možná klika (maximal) – klika, která není indukovaná v žádné jiné klice
- největší klika (maximum) – klika, která obsahuje největší počet vrcholů $\Rightarrow$ každá největší klika je také největší možnou klikou

Nalezení všech největších možných klik je **NP-úplný problém**, protože největší možný počet největších možných klik je exponenciální vzhledem k počtu vrcholů grafu. Pokud nalezneme všechny největší možné kliky, pak nalezení největší kliky lze realizovat výběrem kliky s největším počtem vrcholů.
Jeden z možných způsobů hledání kliky můžeme realizovat postupným zvětšováním už nalezeného úplného podgrafu. Toho dosáhneme rozdělením všech vrcholů do dvou množin, vrcholy obsažené v klice a vrcholy, které mohou být použity pro rozšíření kliky. Při každém rozšíření kliky odebereme z množiny vhodných vrcholů ty, které jsme tímto výběrem



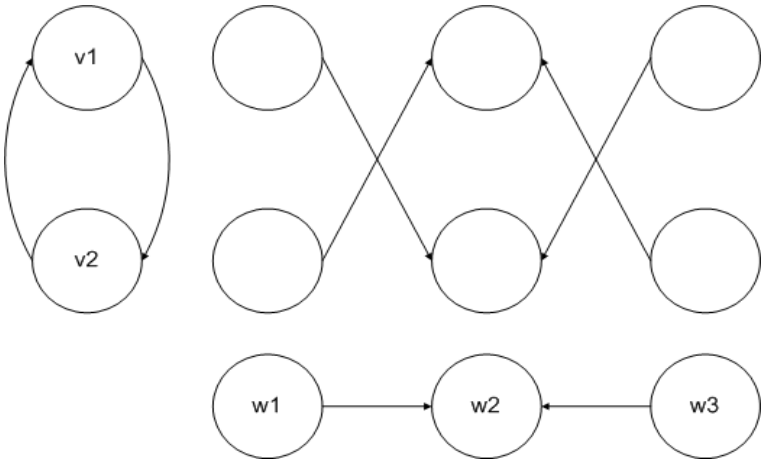
Obrázek 21: Ukázka největší kliky

vyřadily (po přidání těchto vrcholů by podgraf už nebyl úplný). Bližší popis tohoto řešení bude rozebrán dále v kapitole o Backtrackingu.

8.1.9 Modulární produkt

Modulární produkt je graf $G = (V, E)$, který odpovídá **kartézskému součinu dvou grafů** $G_1 \times G_2$ (viz Obrázek 22), kde graf $G_1 = (V_1, E_1)$ a graf $G_2 = (V_2, E_2)$. Vrcholy V grafu G odpovídají $V = V_1 \times V_2$ a hrany modulárního grafu jsou tvořeny $E = \{((v_i, w_i), (v_j, w_j)) \in V \times V \mid v_i \neq v_j, w_i \neq w_j, (v_i, v_j) \in E_1, (w_i, w_j) \in E_2\} \cup \{((v_i, w_i), (v_j, w_j)) \in V \times V \mid v_i \neq v_j, w_i \neq w_j, (v_i, v_j) \notin E_1, (w_i, w_j) \notin E_2\}$. Pokud je tedy mezi vrcholy v_i a v_j hrana, která odpovídá hraně w_i a w_j, pak i modulární graf G bude mít hranu mezi vrcholy (v_i, w_i) a (v_j, w_j), a pokud mezi vrcholy (v_i, v_j) a (w_i, w_j) není hrana, modulární graf G bude mít hranu mezi vrcholy (v_i, w_i) a (v_j, w_j).
Modulární graf už přímo souvisí s porovnáváním dvou grafů a mnoho algoritmů pracuje rovnou s tímto grafem.
Vytvoření modulárního produktu bude mít složitost $O(V_1^2 * V_2^2)$. To přímo odpovídá procházení matice grafu G_1 a grafu G_2. Vytvářet modulární graf lze potom podle daného pseudokódu.

```
1: G = Uzel[V1][V2]; {inicializace modulárního grafu, tvořen uzly, každý uzel ví s kým je spojen}
2: for i pro všechny G1 do
3:   for j pro všechny vazby v i do
4:     for k pro všechny G2 do
5:       for l pro všechny vazby v k do
6:         if G2[k][l] == G1[i][j] a G2[k][l] > 0 then
7:           {hrana existuje a jsou stejné}
8:           vytvoř spojení mezi uzly G[k][i] a G[l][j]
9:         end if
10:        if G2[k][l] == 0 a G1[i][j] == 0 a k ≠ l a i ≠ j then
11:          {hrana neexistuje v obou grafech}
12:          vytvoř spojení mezi uzly G[k][i] a G[l][j]
13:        end if
14:      end for
15:    end for
16:  end for
17: end for
```



Obrázek 22: Modulární produkt

Samotný modulární graf můžeme v aplikaci realizovat dvěma způsoby:

- Uzel modulárního grafu na pozici x a y je realizován jako zásobník, kam dynamicky přidáváme uzly, s kterými je tento uzel spojen (hrany). Toto zpracování více odpovídá obrázku, který je zmíněn výše. Graf neinicializuje žádné zbytečné místo navíc (vhodné pro řídké grafy, i když právě modulární graf často takovým typem grafu není), ale potřebujeme podporu v programovacím jazyce (seznam) či si musíme potřebnou strukturu vytvořit. S tímto grafem se dále lépe pracuje.
- Modulární graf je tvořen maticí (kartézská matice dvou matic původních grafů). Uzel prvního grafu v matici obsahuje počet vrcholů druhého grafu krát řádků nebo sloupců. Takovéto zpracování nepotřebuje žádnou podpůrnou strukturu, ale matice inicializuje i zbytečné hodnoty a celkově se s takovou maticí pracuje hůře (horší realizace budoucích dynamických úprav). Práce s dvourozměrným polem je také často rychlejší než s výše zmíněným zásobníkem.

V pseudokódu je ukázán první způsob. Ten využívá i aplikace, kde je uzel chápán jako objekt třídy Uzel. Objekt této třídy potom obsahuje seznam všech spojení (hran). Hrana obsahuje nejenom odkud a kam směřuje, ale i svou hodnotu (její princip bude vysvětlen v dalších kapitolách práce).

8.2 Procházení grafu

Pro práci s grafem budeme potřebovat algoritmy nutné pro jeho procházení. Výstupem zmíněných algoritmů bude strom, případně les stromů.

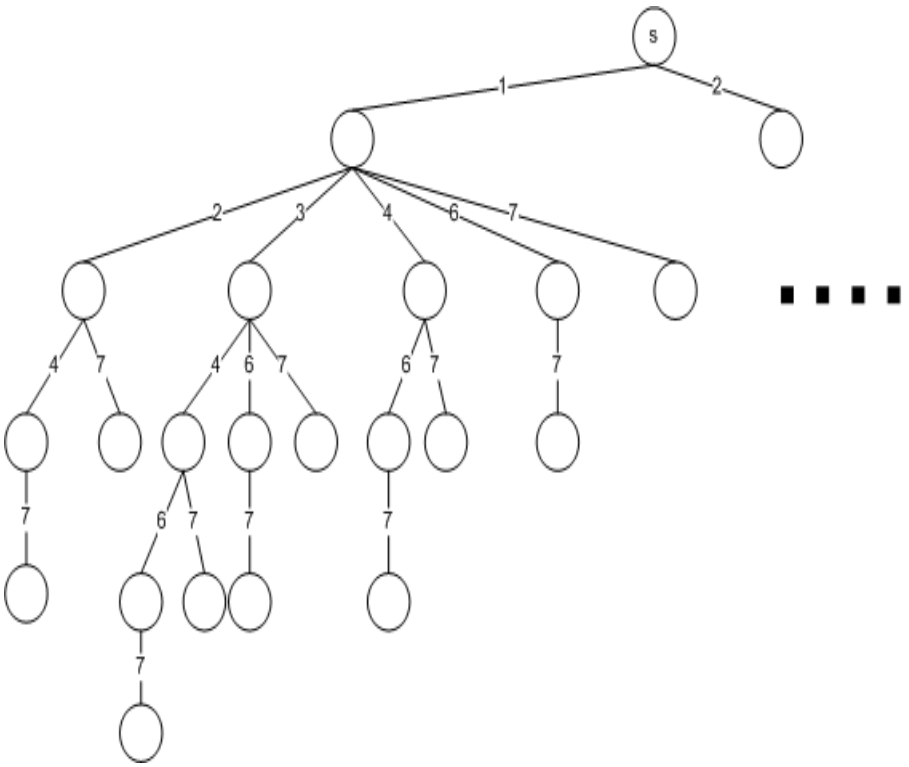
8.2.1 Procházení do hloubky

Procházení do hloubky (DFS z angl. Depth-first search) je algoritmus pro procházení grafu se složitostí O(E+V)–musíme navštívit každý vrchol (po souvislé komponentě putujeme po hranách) a maximální paměťovou náročností O(V)–ta je dána maximální možnou hloubkou stromu.
Tento algoritmus pracuje na principu **LIFO zásobníku** (last in first out). Dochází tedy k rozšiřování výstupního grafu z posledně navštíveného vrcholu. Pokud z daného vrcholu nelze graf už rozšířit, vrací se algoritmus na předposlední navštívený vrchol, kde dále rozšiřuje graf o nenavštívené vrcholy. Pokud opět nemůže graf už dále rozšířit, postupuje podle předešlého postupu, dokud nenavštíví všechny vrcholy (výstupem tohoto algoritmu

je strom).
Abychom mohli procházet i nesouvislé grafy, je nutné před toto procházení předřadit cyklus přes všechny vrcholy, který v konečném důsledku zajistí navštívení všech vrcholů (a výstupem je les stromů–strom pro každou jednu komponentu).

8.2.2 Backtracking (Zpětné vyhledávání)

Procházení grafu založené na principu prohledávání do hloubky v nejhorším případě s exponenciální složitostí. Toto řešení se používá v situacích, kdy není znám žádný rychlejší algoritmus vracející přesné řešení daného problému. Backtracking je **vylepšené prohledávání metodou hrubé síly** a využívá znalost problému, kdy můžeme na základě už zjištěných informací vynechat některé pokusy o kterých víme, že nepovedou ke správnému řešení. „Přísnost“ této funkce (čím více pokusů vynecháme) výrazně ovlivňuje celkovou rychlost algoritmu, v některých případech je i výhodnější snížit přesnost nalezeného řešení, ale zato výrazně zvýšit rychlost výpočtu.
Tato metoda se dá použít například i pro získání největší kliky neorientovaného grafu (viz Obrázek 23). Postup, který byl dříve rozepsán v kapitole o klikách, přesně odpovídá metodě Backtrackingu. Budeme tedy zkoušet všechny možné vrcholy a prodlužovat už nalezený podgraf o vhodné vrcholy (ty vrcholy, které by způsobily neúplnost podgrafu vyřadíme–výrazné zrychlení metody, která by jinak byla pouhou metodou hrubé síly).



Obrázek 23: Ukázka hledání kliky pomocí backtrackingu

Při hledání maximální kliky můžeme backtracking ještě zpřísnit, protože můžeme vyřazovat i všechna řešení, o kterých už víme, že nebudou větší než je nalezená maximální klika (počet vrcholů kliky + počet dostupných vrcholů > počet vrcholů největší kliky).

8.2.3 Procházení do šířky

Procházení do šířky (BFT z angl. Breadth-first traversal) je algoritmus procházení grafu se složitostí O(E+V) a maximální pamět'ovou náročností O(E)–ta je dána tím, že

do paměti ukládáme všechny propojené vrcholy (max V).
Tento algoritmus pracuje na principu **FIFO zásobníku** (first in first out). Dochází tedy k rozšiřování výstupního grafu z prvního navštíveného vrcholu (potřebujeme tedy dostatečně velký zásobník pro ukládání dalších navštívených vrcholů, ke kterým se budeme vracet).
Zde si vrcholy označujeme stejně tak, jako v případě algoritmu DFS, čímž rozlišujeme navštívené a nenavštívené vrcholy.
Zatímco DFS využíváme pokud chceme najít co největší podgraf (maximální klika, atd.–v takovém případě potom můžeme díky backtrackingu omezit budoucí pokusy), BFS použijeme pokud chceme najít jedno odpovídající řešení (v kolikátém kroku nejdříve dojde graf do stavu y ze stavu x – nalezené řešení se rovná konci výpočtu). BFS sestavuje postupně celé úrovně výstupního stromu.

8.3 Specifikace grafového problému

Se znalostí grafů a dříve zmíněných termínů se nyní můžeme pokusit definovat problém vyhledávání tak, aby odpovídal přesně problému v oblasti teorie grafů. Problém si rozebereme v několika krocích, jejichž spojením dosáhneme výsledné definice.

1. Budeme hledat společný podgraf dvou grafů $G_1 = (V_1, E_1, We_1, Wv_1)$ a $G_2 = (V_2, E_2, We_2, Wv_2)$ a to takový, že výsledný graf G bude tvořen vrcholy $V \subseteq V_1 \times V_2$ a hranami E spojující tyto vrcholy, pro které platí $We_1 = We_2$. Výsledný podgraf tedy bude obsahovat hrany a vrcholy, které jsou oběma grafům společné. Společný podgraf budeme hledat, protože ho můžeme následně použít jako měřítko podobnosti.

2. Z nalezených společných podgrafů G nás bude zajímat ten maximální ⇒ maximální shoda.

3. Musíme si uvědomit, že podgraf je isomorfní (různé namapování vrcholů V_1 na V_2). Pokud tedy porovnáváme graf G_1 a G_2, pak o nich můžeme prohlásit, že jsou shodné, jestliže ověříme, že graf G_1 je isomorfní ke grafu G_2 a naopak. To je dáno tím, že posloupnosti vrcholů V_1 a V_2 a příslušných hran mohou být jinak namapovány (resp. přejmenovány). Vrchol $v_1 \in V_1$ tedy může odpovídat libovolnému vrcholu z množiny vrcholů V_2. Měřítkem shodnosti je tedy isomorfismus a při hledání společného podgrafu musíme hledat společný isomorfní podgraf.

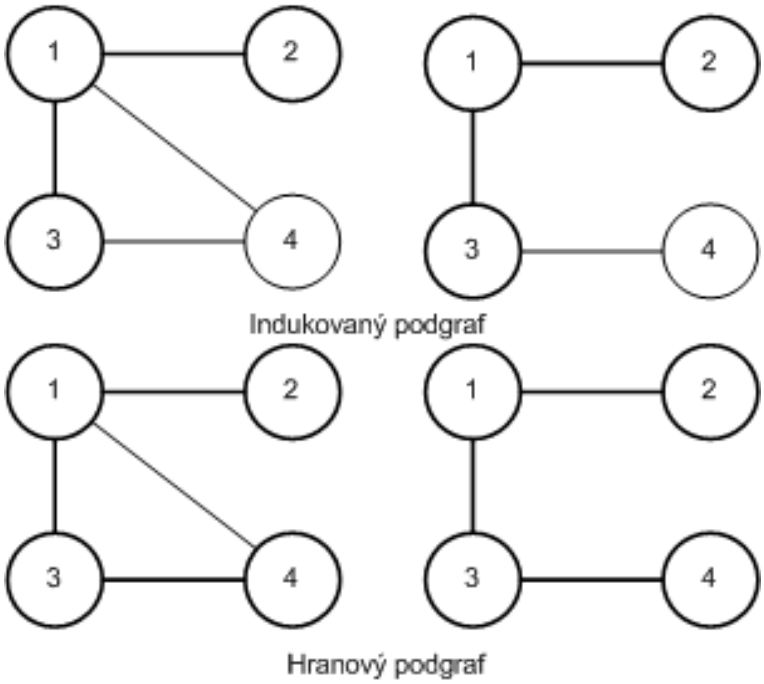
4. Budeme porovnávat dva orientované, nesouvislé grafy (neobsahují smyčky ani vícenásobné hrany).

Pokud shrneme všechny tyto informace do jediné definice, můžeme náš problém označit za **hledání maximálního společného isomorfního podgrafu** (maximal common subgraph isomorphism), pro který platí, že maximální společný isomorfní podgraf grafu G_1 do G_2 je struktura (S_1, S_2, M), kde S_1 je podgraf G_1 a S_2 je podgraf grafu G_2 a M je isomorfní graf podgrafu S_1 do podgrafu S_2.[17]
Rozlišujeme dva druhy hledání maximálního společného isomorfního podgrafu (viz Obrázek 24):

- maximální společný indukovaný podgraf (MCIS) – výsledný podgraf je indukovaný (obsahuje všechny hrany na podmnožině vrcholů podgrafu původního grafu ⇒ to často může vést ke snížení velikosti výsledného podgrafu)

- maximální společný hranový podgraf (MCES) – výsledný podgraf nemusí být indukovaný (často mnohem přesnější řešení, ale toto hledání nelze například zredukovat na hledání maximální kliky modulárního grafu)

Některé algoritmy jsou vhodné pro hledání indukovaného i hranového podgrafu (McGregor), další algoritmy vrací pouze indukovaný graf (to je často dáno již zmíněnou redukcí na hledání maximální kliky), příkladem takového algoritmu je například Valientův algoritmus.
Většina algoritmů se dnes soustředí na určování MCIS, z toho důvodu je často potřeba graf MCIS transformovat na MCES.[12]



Obrázek 24: Rozdíl mezi maximálním společným indukovaným a hranovým podgrafem

8.3.1 Redukce na problém hledání maximální kliky

Maximální společný indukovaný podgraf lze redukovat na p**roblém hledání maximální kliky**. I když je hledání kliky také NP-úplný problém, můžeme zde uplatnit větší skupinu algoritmů (od backtrackingu až po různé další algoritmy–např. Bron-Kerboschův algoritmus).
Pro zmíněnou redukci použijeme modulární produkt dvou grafů. Nechť existují $G_1 = (V_1, E_1)$ a $G_2 = (V_2, E_2)$ a graf $C = (V, E)$, který je úplným podgrafem grafu $G_1 \times G_2$ (odpovídá kartézskému součinu grafů, resp. modulárnímu produktu). Projekce grafu C do grafu G_1 je podgraf grafu G_1 indukovaný na vrcholech $v \in V_1$, tak že $(v, w) \in V$, kde $w \in V_2$. Pro druhý graf potom platí obrácená ekvivalence. Projekce grafu C do grafu G_2 je podgraf grafu G_2 indukovaný na vrcholech $w \in V_2$, tak, že $(v, w) \in V$, kde $v \in V_1$.
Modulární produkt vyjadřuje možné namapování vrcholů $v \in V_1$ a $w \in V_2$. Pokud využijeme už známou definici modulárního produktu a poznatek, že mezi vrcholy (v_i, w_i) a (v_j, w_j) je hrana, pokud je hrana mezi vrcholy v_i a v_j, která odpovídá hraně w_i a w_j a nebo pokud mezi vrcholy (v_i, v_j) a (w_i, w_j) hrana není. Z toho vyplývá, že hrana mezi vrcholy (v_i, w_i) a (v_j, w_j) nebude v případech, kde se hrany vrcholů (v_i, v_j) a (w_i, w_j) nerovnají, případně mezi vrcholy (v_i, v_j) existuje hrana, zatímco mezi vrcholy (w_i, w_j) ne a naopak. Nalezený podgraf C tedy musí být úplný aby na grafech G_1 a G_2 odpovídal projekci indukovaného podgrafu $\Rightarrow$ na modulárním grafu budeme hledat kliku (platí, čím větší klika, tím větší společný podgraf grafů G_1 a G_2).
Modulární graf by měl být neorientovaný z důvodu hledání kliky na tomto grafu. U nás ale budou vstupem dva orientované grafy, jejichž modulární produkt bude také orientován.

Transformaci z orientovaného grafu do neorientovaného provedeme tak, že pokud vede hrana z (v_i, w_i) do (v_j, w_j), potom musí vést i opačně. Pokud modulární produkt takovou hranu neobsahuje, jednoduše ji přidáme. To si můžeme dovolit proto, že budeme na tomto grafu hledat podgraf (kliku) a díky vlastnostem modulárního produktu $\Rightarrow$ jestliže výsledný podgraf obsahuje (v_i, w_i) i (v_j, w_j), nezáleží nám, zda-li mezi nimi existuje neorientovaná nebo orientovaná hrana (hrany zde vyjadřují možné namapování daných párů vrcholů). Pokud nechceme ztratit informaci, jestli před touto transformací vedla hrana z vrcholu (v_i, w_i) do (v_j, w_j), případně opačně, můžeme si jednotlivé neorientované hrany ohodnotit. To se může hodit, například chceme-li najít nejen kliku s největším počtem vrcholů, ale i největším množstvím hran. Příkladem použitého hodnocení může být:

- hodnota hrany 0 – mezi vrcholy $(v_i, v_j) \in V_1$ a $(w_i, w_j) \in V_2$ není hrana $\Rightarrow$ nalezený podgraf se nezvětší o žádnou hranu
- hodnota hrany 1 – mezi vrcholy modulárního grafu (v_i, w_i) a (v_j, w_j) byla před transformací jedna orientovaná hrana $\Rightarrow$ aspoň u jednoho grafů mezi vrcholy (v_i, v_j) a (w_i, w_j) vedla jedna orientovaná hrana (u druhého mohla vést jedna orientovaná, případně neorientovaná hrana – neorientovaná hrana je chápana jako dvě orientované hrany)
- hodnota hrany 2 – mezi vrcholy modulárního grafu (v_i, w_i) a (v_j, w_j) byly před transformací dvě orientované hrany

Toto hodnocení nám dovoluje vybrat nejen maximální kliku s největším možným počtem vrcholů, ale i kliku, u které bude počat hran na grafech G_1 a G_2 největší.

8.4 Přehled vhodných algoritmů

Volba vhodného algoritmu výrazně ovlivňuje nejen výsledek samotného vyhledávání (MCIS nebo MCES), ale i jeho rychlost, případně průběh výpočtu a realizace vyhledávání v aplikaci.
Víme, že isomorfismus i maximální společný podgraf dvou grafů lze určit použitím hrubé síly. Postupně budeme metodou prohledávání do hloubky zkoušet všechna řešení, dokud nenalezneme to nejlepší. Na tyto myšlenky navazuje James J. McGregorův algoritmus. Ten využívá metody backtrackingu, v každém kroku tedy odstraňuje nevhodná řešení, což výrazně urychluje celý výpočet.
Dalším algoritmem, který řeší tento problém jinak než McGregorův, je Durand-Pasari algoritmus. Zde je využita již zmíněná redukce problému hledání maximálního společného indukovaného podgrafu na problém hledání maximální kliky nebo Balas-Yu algoritmus, který využívá barvení grafu.[16]
Jedním z použitých algoritmů, který vychází z Valientovi knihy, je Valientův algoritmus, který vyhledává MCIS na modulárním produktu dvou grafů.[11]
Algoritmů zabývajících se problémem porovnávání dvou grafů existuje celá řada a není v možnostech ani smyslem této práce zmínit je všechny. Proto si vystačíme s tímto krátkým přehledem, na kterém bude detailněji popsán celý postup porovnávání.
Všechny zde zmíněné algoritmy jsou označovány jako **přesné**. Toto označení definuje, že výsledkem algoritmu je vždy přesné řešení daného problému. Cenou za tuto přesnost je v nejhorším případě exponenciální složitost výpočtu.[14] Vzhledem k tomu, že naše aplikace je míněna především na porovnávání menších softwarových struktur (zhruba okolo 10 statických prvků$\Rightarrow$ graf o maximálně 10 vrcholech), jsou použity právě přesné algoritmy, které na grafech o těchto velikostech v průměru dosahují přijatelných časových složitostí. Tyto algoritmy jsou použitelné i pro srovnávání struktur proteinů.[13]

8.4.1 McGregorův algoritmus

Tento algoritmus je založen na metodě prohledávání do hloubky (resp. na metodě backtrackingu). V každém kroku algoritmu se vybere dvojice vrcholů v a w, kde $v \in V_1$ a $w \in V_2$ a zkoumá se, jestli mohou zvětšit už nalezený největší společný podgraf. Pakliže je možno zvětšit nalezený podgraf, zvětšíme podgraf a pokračujeme dále v porovnávání až dokud nevybereme postupně všechny vrcholy. Hloubka výsledného stromu nikdy nemůže překročit velikost minimálního grafu.
Vzhledem k použité metodě prohledávání do hloubky je každé rozšíření podgrafu chápáno jako možné větvení na výsledném stromě. Pomocí tohoto větvení se vždy vracíme zpět a snažíme se najít ještě lepší řešení. V každém kroku prohledávání může dojít k vynechání některých porovnávání na základě omezujících podmínek. Takovými podmínkami se rozumí stav o kterém víme, že i při dalším rozšiřování podgrafu jsme v minulosti nalezli už stejně lepší řešení.[15]

```
1: {McGregor(s): kde s je nalezený strom}
2: while vyber vrcholy v,w do
3:    if lze rozšířit s o v,w then
4:       rozšiř výsledný podgraf
5:       if s > maxpodgraf then
6:          maxpodgraf = s
7:       end if
8:       if neprošli jsme všechny vrcholy && !omezeni(s) then
9:          rekurze McGregor(s)
10:      end if
11:   end if
12: end while
```

Pokud platí $N_1 \leq N_2$, kde N_1 je počet vrcholů grafu G_1 a N_2 je počet vrcholů grafu G_2. Potom bude mít algoritmus nejhorší možnou složitost $O(\frac{(N_2+1)!}{N_2-N_1+1}!)$. Tato složitost je dána tím, že algoritmus bude muset projít $(N_2 + 1)$ uzlů na první úrovni, N_2 uzlů na druhé úrovni (potomci předchozího uzlu⇒pro celkový počet musíme tyto úrovně vynásobit) až do $(N_2 - N_1 + 2)$ uzlů na poslední N_1 úrovni. Paměťová náročnost je pouze $O(N_1)$ ⇒ nutnost backtrackingu do maximální hloubky výsledného stromu, tedy N_1.
McGregorův algoritmus dovoluje vyhledávat MCIS i MCES a to na základě podmínky, kterou si přesněji stanovíme pro přidávání vrcholů v a w do výsledného podgrafu s. Je nutno si uvědomit, že vyhledávání MCES bude často výpočetně náročnější díky neexistenci omezující podmínky na indukčnost podgrafu.

8.4.2 Valientův algoritmus

Valientův algoritmus využívá principu algoritmu Durand-Pasari. K hledání MCIS využívá redukci na problém hledání maximální kliky. Toto hledání je realizováno za pomoci backtrackingu na modulárním produktu dvou grafů.
Tvorba modulárního produktu byla zmíněna již dříve, a proto ji zde nebudu už znova rozebírat. Důležité v tomto bodě je zmínit a ukázat, jak je v aplikaci realizována metoda backtrackingu pro samotné nalezení maximální kliky.

```
1: {maxKlika(vysledek, dalsiBody)}
2: if vysledek > maxVysledek then
3:    maxVysledek=vysledek
4: end if
5: if dalsiBody nejsou prázdné then
6:    for i pro všechny dalsiBody do
7:       odstraň i z dalsiBody{Nebudeme zbytečně prohledávat některá řešení dvakrát}
```

```
8:      dalsiBody2
9:      for j pro všechny cesty v i do
10:       if dalsiBody obsahuje i then
11:         dalsiBody2 přidej i {Přidáme pouze vrcholy, tak aby byl podgraf úplný}
12:       end if
13:     end for
14:     if maxVysledek < vysledek + dalsiBody2 then
15:       {Hledáme pouze maximální kliku–tím odstraníme hodně zbytečných prohledávání}
16:       maxKlika(vysledek,dalsiBody2)
17:     end if
18:   end for
19: end if
```

Z ukázky pseudokódu je patrné, že v každé rekurzi zmíněné metody, si algoritmus udržuje přehled vrcholů o které může nalezený výsledek (výsledek odpovídá nalezenému podgrafu) rozšířit. Ze seznamu jsou v každé rekurzi odstraněny vrcholy, které by po přidání porušili výslednou úplnost podgrafu. Takováto řešení vůbec nemusíme procházet a tím výrazně zrychlit vyhledávání.
Použitý algoritmus vyhledává pouze maximální kliku. Ve chvíli kdy už o nalezeném řešení můžeme prohlásit, že už nebude větší než maximální (nalezený podgraf + všechny dostupné vrcholy o které ho můžeme rozšířit < maximální klika), je tato část výsledného stromu ukončena.
Složitost tohoto algoritmu, v nejhorším případě, kdy oba grafy budou úplné nebo žádný z vrcholů v obou grafech nebude spojen hranou, bude $O(\frac{(N_2+1)!}{N_2-N_1+1}!)$, kde N_1 je počet vrcholů grafu G_1 a N_2 je počet vrcholů grafu G_2. Paměťová náročnost algoritmu potom bude $O(N_1 * N_2) \Rightarrow$ ta je dána nutností tvorby modulárního produktu. Pokud platí, že $N_1 = N_2 = N$, můžeme složitost redukovat na vztah $O(N * N!)$.

8.4.3 Bron-Kerbosch algoritmus

Bron-Kerboschův algoritmus je algoritmus pro hledání klik na grafu, s maximální složitostí $O(3^{n/3})$, kde n je počet vrcholů a složitost $3^{n/3}$ symbolizuje maximální počet všech maximálních možných klik v n-grafu. Tento algoritmus výrazně zrychluje hledání samotné kliky na grafu například oproti backtrackingu u Valientova algoritmu.
Pokud tento algoritmus aplikujeme na modulární produkt dvou grafů, potom ho můžeme použít i pro řešení našeho porovnávacího problému a najít maximální společný podgraf. Je důležité si uvědomit, že například dva grafy, každý o 10 vrcholech, budou mít výsledný modulární graf o 100 vrcholech. V nejhorším případě by proto bylo podle zmíněné složitosti potřeba několik biliónů rekurzí pro nalezení všech klik. Abychom vznik takovéto situace omezili, budeme vyhledávat pouze největší kliku grafu, která je pro nás podstatná.[18][19]
Bron-Kerboschův algoritmus rozděluje vrcholy grafu do několika skupin:

- kandidáti – o tyto vrcholy lze rozšířit nalezený podgraf (kliku) a toto řešení ještě nebylo prozkoumáno
- not (ne) – vrcholy, které by kliku mohli rozšířit, ale jejich výsledek už jsme prozkoumali
- výsledek – vrcholy vybraného podgrafu

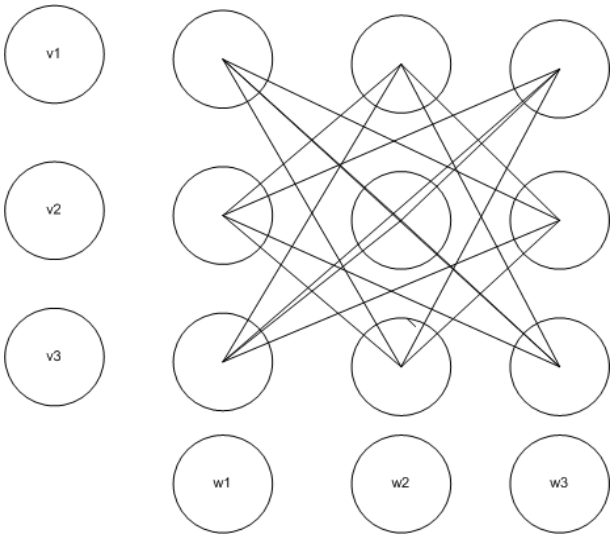
Nalezená klika je maximální pouze pokud neobsahuje žádné vrcholy ve skupinách kandidáti a not.

```
1: {bronKerbosch(vysledek, kandidati, not)} {Najdeme vhodného kandidáta - pokud budeme hledat pouze největší kliku, potom nám tento výběr kandidáta počet rekurzí příliš nesníží}
2: for i pro všechny kandidáty do
3:    for j pro všechny not do
4:       nespojeni++
5:    end for
6:    if nespojeni<max then
7:       vyber kandidáta, který není spojen s min not
8:    end if
9: end for
10: for i pro všechny kandidáty do
11:    setřiď prvky, abychom začali s vybraným kandidátem
12:    kandidatiNew=smaž kandidáty nespojené s i
13:    notNew=smaž not vrcholy nespojené s i
14:    přidej i do vysledek
15:    if kandidatiNew je prázdná && notNew je prázdná then
16:       if vysledek>nejlepsi then
17:          nejlepsi=vysledek {uložení největší kliky}
18:       end if
19:    else
20:       if notNew je prázdná then
21:          bronKerbosch(vysledek, kandidatiNew, notNew) {zajímají nás pouze maximální kliky}
22:       end if
23:    end if
24:    smaz i z vysledek {musíme smazat kvůli rekurzi}
25:    přidej i do not {už prozkoumané řešení}
26: end for
```

Algoritmus v každém kroku vybírá nejvhodnějšího kandidáta, podle pravidla: vybereme vrchol, který není spojen s vrcholem ze skupiny not, který má nejméně spojení s kandidáty. Vybraný kandidát je přidán do řešení a jsou znova spočítány skupiny kandidátů a not. Při návratu z rekurze jsou tyto skupiny obnoveny do minulého stavu a vybraný kandidát je přidán do not skupiny. Algoritmus končí pokud je počet dalších kandidátů nulový a nebo pokud existuje vrchol v not, který je spojen se všemi zbylými kandidáty.

8.4.4 Realizované úpravy algoritmů

Při implementaci algoritmů využívajících modulární produkt a hledání největší kliky narážíme na několik problémů.
I když budou oba porovnávané grafy poměrně malé, pokud nastane jeden ze zmíněných nejhorších případů (viz Obrázek 25), potom bude doba výpočtu poměrně dlouhá. Příkladem nejhoršího případu může být když porovnávané grafy budou úplné a nebo naopak bez hran⇒ i modulární graf bude úplný. Abychom minimalizovali dobu trvání výpočtu, můžeme softwarově omezit uživatele a velikost grafu, který bude moci nakreslit. Toto řešení je správné a od určitého bodu nezbytné, ale tímto způsobem nezrychlíme samotný algoritmus.
Případy kdy budou oba grafy úplné a jejich vrcholy nebudou pojmenované jdou řešit poměrně jednoduše. Výsledný maximální společný podgraf C, bude graf G_1, který bude na grafu G_2 indukován, pokud platí $G_1 \leq G_2$. V aplikaci, kdy jsou vstupem dva orientované grafy je ovšem tato situace poměrně řídká.
Opačným extrémním případem je stav, kdy ani jeden z grafů nebude obsahovat hranu.



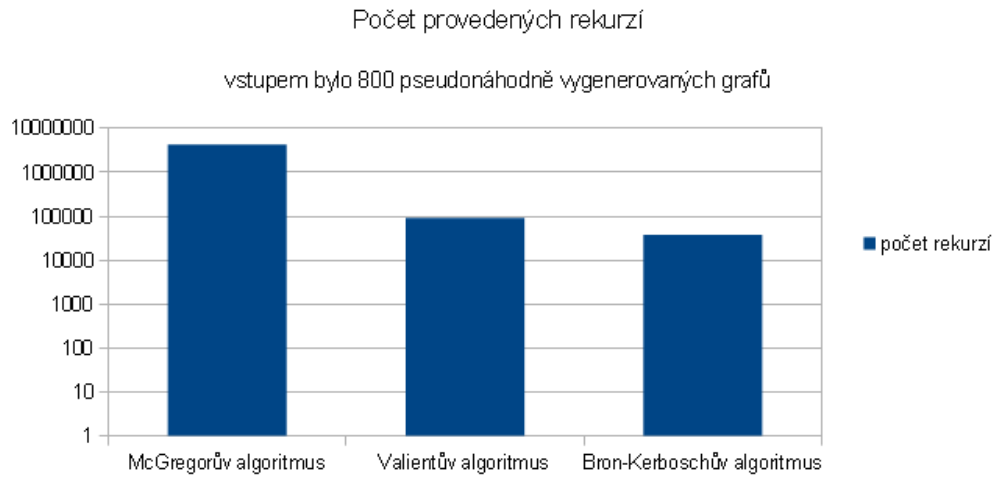
Obrázek 25: Příklad nejhoršího případu modulárního grafu

Víme ovšem, že osamocené vrcholy grafu G_1 můžeme namapovat na vrcholy grafu G_2, bez toho aniž bychom nějak ovlivnili zbytek podgrafu. Toto namapování si tedy můžeme udělat mimo algoritmus a v modulárním produktu grafů ponechat jen ty vrcholy, které nějakou hranu obsahují.
Je důležité si uvědomit, že každá hrana grafu G_1 neobsažená v grafu G_2, výrazně snižuje náročnost celkového výpočtu.
Dalším problémem je, jak vlastně vyhodnocovat podobnost dvou grafů na základě nalezené maximální kliky. Maximální klika je ta, která obsahuje největší počet vrcholů a algoritmus po nalezení této kliky, všechny menší a stejné kliky zahazuje. Tato skutečnost celkový výpočet velmi zrychluje, ale také jednoznačně pro nás zhoršuje nejvýhodnější řešení.
Za toto nejvýhodnější řešení se dá označit podgraf, který obsahuje nejen maximální počet vrcholů, tak jak nám jej vrací maximální klika, ale i maximální počet hran. Až na základě těchto čísel vrátíme uživatelovi procento shody. Může proto nastat situace, kdy vrácený podgraf bude obsahovat čtyři vrcholy, ale jen dvě hrany, a protože tato klika bude nalezena dříve než následující podgraf obsahující čtyři vrcholy a sedm hran, bude druhé řešení zahozeno (resp. algoritmus ho ani nebude na základě backtrackingu zkoušet). Z tohoto příkladu je patrné, že druhý případ by byl oznámkován větší procentuální shodou a tím pádem přesnějším řešením tohoto problému.
Zde stojíme opravdu před závažným problémem, jehož řešením by muselo být výrazné potlačení omezující podmínky a dovolit algoritmu vyhledávat i řešení o menším a stejném počtu vrcholů, než je maximální. Z nich potom počítat procentuální shodu a vybrat tu největší.
Já jsem se rozhodl toto řešení nepoužít v celé jeho šíři, resp. dovoluji algoritmu stále pracovat i s podgrafy o kterých už vím dopředu, že nebudou větší než je maximální klika, ovšem ani že nebudou menšími (v pseudokódu je v závěrečné omezující podmínce přidáno ještě rovnítko). Modulární produkt má všechny hrany ohodnoceny a v průběhu sestavování podgrafu je počítán i součet hran. Pokud naleznu podgraf, který má stejný počet vrcholů a větší počet hran, potom tímto podgrafem nahradím dosud maximální uložený podgraf. Toto řešení je ovšem zjednodušením celého problému a v některých případech nemusí vracet nejideálnější řešení. Například pokud bude jako největší společný podgraf vrácen podgraf o pěti vrcholech a třech hranách a přitom bude jedním z dalších možných podgrafů podgraf o sice čtyřech vrcholech, ale osmi hranách. Druhý případ by samozřejmě vyšel procentuálně lépe a byl lepším řešením. Museli bychom ovšem procházet mnohoná-

sobně více možných řešení. Tato heuristika je tedy použita z důvodů velkého zrychlení a je při ní kladen větší důraz na maximální počet vrcholů než hran.

8.5 Reálná složitost použitých algoritmů

Pro výběr algoritmu hraje důležitou roli jeho složitost a tím daná časová náročnost výpočtu. Abychom tento výběr zpřesnili, budeme společně s nejhorší možnou složitostí danou $O(n)$ brát v potaz i reálnou náročnost (viz Obrázek 26).
Pro určení této náročnosti byl každý z použitých algoritmů testován oproti několika stovkám pseudonáhodných grafů i několika určených grafů zjišťující funkčnost pro daný problém (například porovnávání dvou nesouvislých grafů).
Vzhledem k některým realizovaným úpravám na algoritmech nemůžeme exaktně prohlásit, že McGregorův algoritmus bude pomalejší než Valientův. V realizované aplikaci s danými úpravami ovšem můžeme reálně prohlásit, že McGregorův algoritmus bude pomalejší než Valientův algoritmus. A použití Bron-Kerboschova algoritmu pro hledání maximální kliky výrazně urychlí její hledání oproti backtrackingu (toto zrychlení je nicméně patrné už ze zmíněné složitosti obou algoritmů).
Pro porovnávání grafů v aplikaci se tedy jako nejvýhodnější jeví hledání největší kliky na modulárním produktu dvou grafů pomocí algoritmu Bron-Kerbosch. Toto porovnávání je nicméně použitelné pouze pro hledání MCIS. Pro hledání MCES by stačilo vhodně upravit McGregorův algoritmus, resp. odstranit některá omezení na výslednou indukčnost podgrafu. Takovéto vyhledávání by nicméně bylo časově nejnáročnější ze všech použitých řešení.



Obrázek 26: Srovnání algoritmů při porovnávání dvou grafů o pěti vrcholech

9 Realizace navrženého systému

Zatímco v předchozí kapitole Návrh aplikace jsem popisoval postupný návrh a vnitřní strukturu aplikace, zde budu mluvit už o příslušné realizaci v jazyce Java. Tento jazyk byl zvolen pro svou multiplatformnost a tím danou přenositelnost.
Protože třídy a jejich vazby byly realizovány podle předchozího návrhu, nebudu zde již rozebírat, co která třída dělá. a proč jsou třídy v tomto spojení, a více se zaměřím na aplikaci z pohledu uživatele. Ukážu zde její možnosti a výslednou grafickou podobu formou obrázku. Uvedu občas i použitý zdrojový kód.

Složka s jar souborem (tedy se spustitelným souborem aplikace) obsahuje ještě několik dalších složek a souborů nutných pro správné fungování aplikace.

- lib – Složka obsahující použité externí knihovny, v našem případě obsahuje pouze knihovnu databáze db4o. Tato složka, resp. soubor v ní, je nepostradatelný pro fungování aplikace.
- ikony – Obsahuje obrázky ve formátu png. Ty jsou použity pro tlačítka kreslených entit. Pokud aplikaci nedodáme tuto složku s příslušným souborem, nebude tlačítko obsahovat jako pozadí daný obrázek, a tím uživateli znepříjemní další manipulaci při kreslení. Při návrhu celé aplikace byl kladen důraz na co nejjednodušší rozšiřitelnost aplikace o jednotlivé položky kreslícího panelu a jejich příslušné obrázky.
- data – Výchozí složka pro ukládání a načítání dat. Sem je standardně nasměrováno okno pro výběr souboru pro načtení/uložení. Tato složka je pro aplikaci postradatelná.
- napoveda – Název složky obsahuje htm soubory použité k zobrazení nápovědy. Bez této složky nebude aplikace obsahovat žádnou nápovědu.
  Soubory htm jsou z této složky dynamicky načítány do aplikace. To velmi usnadňuje rozšiřitelnost celé nápovědy. Formát htm je použit pro lepší vizuální znázornění daného textu.
- setup – Soubor setup ve formátu txt obsahuje počáteční nastavení aplikace. Tento soubor je postradatelný, protože aplikace si umí vytvořit nový výchozí setup soubor.

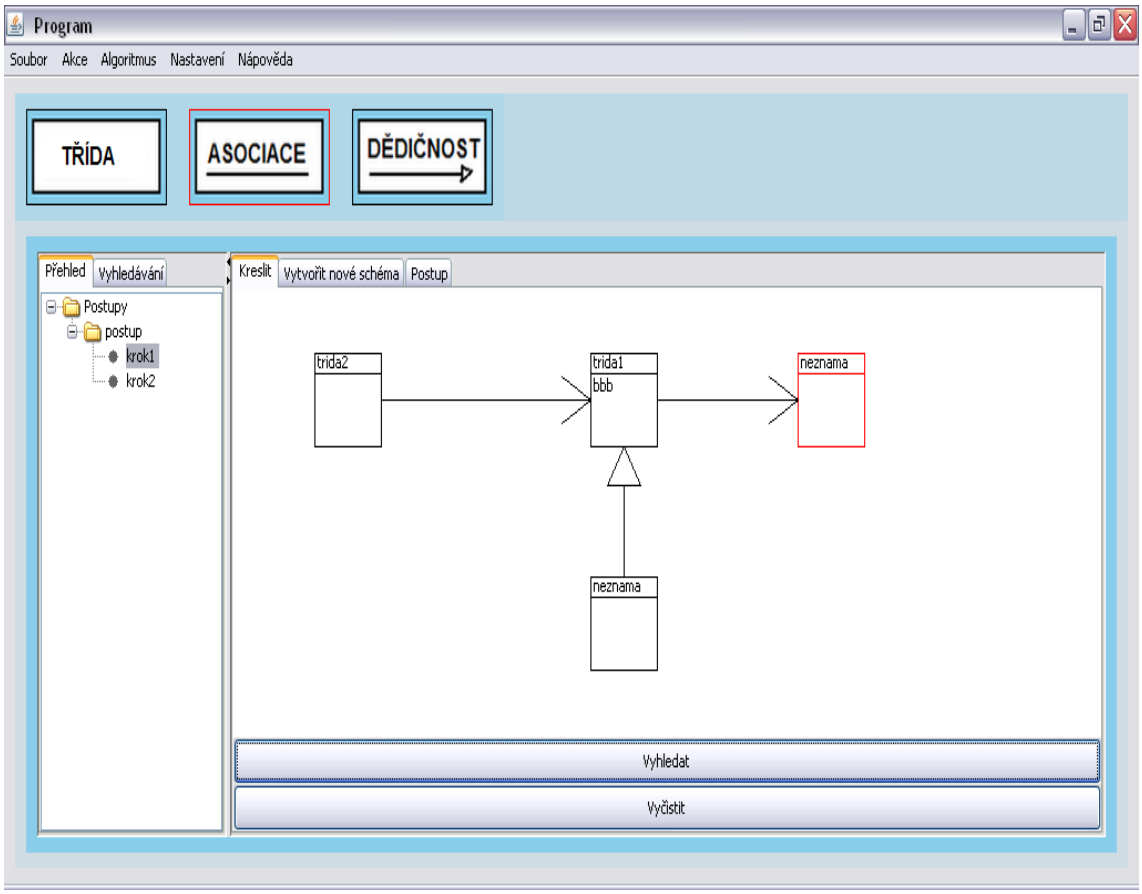
9.1 Grafické uživatelské rozhraní (GUI)

Grafické uživatelské rozhraní bylo v zásadě realizováno podle drátového modelu vytvořeného během návrhu s přihlédnutím k možnostem jazyku Java (viz Obrázek 27).[22] Menu aplikace obsahuje všechny nutné položky pro správu aplikace.

- Soubor – správa databáze (načítání a ukládání dat, vytváření nových databází), většina těchto operací pracuje v jiném vláknu aplikace, aby uživatel mohl plynule pokračovat v práci a nebo při načítání dat šel zobrazit jeho postupný průběh
- Akce – zde jsou všechny dostupné akce pro správu jednotlivých postupů a jejich panelů
- Algoritmus – ruční výběr vhodného algoritmu pomocí přepínače ⇒ vždy můžeme využívat pouze jeden vyhledávací algoritmus. Toto podmenu je vytvářeno dynamicky, abychom ho nemuseli upravovat při rozšíření systému o nové vyhledávací algoritmy.
- Nastavení – umožňuje měnit nastavení aplikace a Setup souboru. Toto podmenu je vytvářeno dynamicky na základě obsahu Setup objektu ⇒ objekt třídy Setup je Jedináček.
- Nápověda – přístup k oknu nápovědy a nebo dalším dodatečným informacím o aplikaci

Pod menu je umístěn panel s položkami možných kreslených entit. Jako pozadí pro jednotlivé položky je použit obrázek ze složky ikony. Pro změnu vykreslování tlačítka je přepsána metoda:

```
1 protected void paintComponent(Graphics g) {}
```



Obrázek 27: Grafické uživatelské rozhraní

Dále je grafické rozhraní rozděleno pomocí komponenty JSplitPane na dva kontejnery – přehled a obsah. Komponenta JSplitPane dovoluje dynamicky měnit velikost obou kontejnerů.
Část přehledu obsahuje komponentu JTabbedPane, která umožňuje přepínání mezi několika kontejnery a vykreslování pouze aktivního kontejneru. Do této komponenty jsou vloženy další dvě komponenty JTree (resp. její potomci). První komponenta JTree obsahuje přehled dat v databázi, druhá potom výsledky vyhledávání. Tyto komponenty jsou mezi sebou synchronizované, aby změna jedné ovlivnila i druhou.
Druhá část obsahu je tvořena také komponentou JTabbedPane. Do ní je vložen vyhledávací panel kreslení, formulář pro vytvoření nového postupu a přehled vybraného postupu (tento přehled obsahuje další JTabbedPane pro jednotlivé kroky postupu). Na vyhledávací panel nebo kreslící panely jednotlivých kroků může uživatel už kreslit. Změna kreslící strategie je řešena změnou aktivní komponenty na JTabbedPane obsahu (přidání listenera).

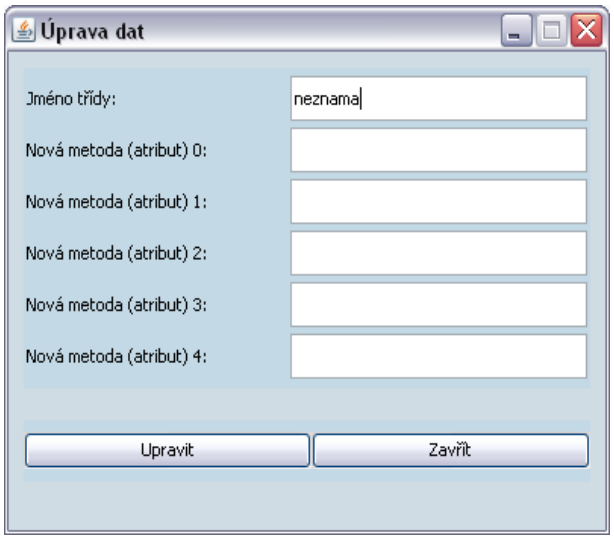
```
1 public void stateChanged(ChangeEvent e) {}
```

Neméně důležitou částí grafického rozhraní je jeho zvolená barva. V budoucnu se může stát, že budeme chtít změnit barvu aplikace. Z toho důvodu jsou atributy barev deklarovány pouze v jednom objektu a dalším objektům jsou předávány jako reference. Zatím systém neumožňuje samotnému uživateli měnit barvu aplikace, ovšem s tímto opatřením by tento možný budoucí požadavek měl být lehce realizovatelný.
Při spuštění nebude aplikace obsahovat část přehledu, ani obsahu. To je dáno tím, že tyto části pracují přímo s daty v databázi a dokud uživatel nevybere soubor, který si přeje načíst, nebudou tyto části vůbec vytvořeny. Po výběru souboru s daty (při práci s daty je potřeba odchytávat výjimky a informovat uživatele pomocí návrhového vzoru Pozorovatel)

se zobrazí uživateli indikátor průběhu (tento indikátor neukazuje kolik dat už je načteno, pouze informuje, že jsou data načítána). Indikátor je vytvořen za pomocí Java komponenty JProgressBar. Po načtení všech dat bude indikátor z plochy odstraněn a nahrazen novým obsahem z databáze.
Převážná většina komponent je na panely umístěna za pomocí GridBagLayoutu a příslušných GridBagConstraints (potřebná omezení). Toto rozložení nám dovoluje dynamicky reagovat na změnu velikosti okna a určovat procentuálně, řádkem i sloupcem, umístění příslušné komponenty na ploše.

9.2 Kreslení

Kreslení je v aplikaci umožněno pouze na kreslící panely. Vazby se mezi statickými prvky vykreslují podle předchozího návrhu. Samotný panel obsahuje několik listenerů, kteří umožňují obsloužit všechny uživatelovi požadavky.
Kreslení začíná výběrem prvku z panelu dostupných entit. Aktivní prvek změní barvu svého ohraničení na červenou. Opětovným kliknutím na prvek dojde k deaktivaci tohoto prvku. Tato změna je navázána na listener akce příslušného tlačítka. V celém panelu může být aktivní pouze jedna entita.
Pro změnu velikosti statických prvků je potřeba držet klávesu r (resize). Události na klávesnici jsou zachytávány pomocí listeneru KeyEventPostProcessor a dále filtrovány.
Na plátně je přímo zachytáváno kliknutí. Pokud klikneme mimo už nakreslený prvek, dojde k vykreslení aktivního prvku (vytvoření nového objektu a přidání do zásobníku). Jestliže klikneme už na existující prvek, dojde k aktivování tohoto prvku (změna barvy). Tyto průniky jsou ošetřeny pomocí metody intersects třídy Shape. Protože je počítání těchto průniků velmi náročné, tato metoda úlohu zjednodušuje na průnik specifických 2D obdélníků. Z toho plyne, že zatímco hranaté prvky půjdou označit velmi snadno a přesně, vazby i při přesném kliknutí přímo na ně nemusí být označeny.



Obrázek 28: Úprava vlastností třídy

Dvojitým kliknutím na už existující prvek dojde k otevření dalšího okna pro úpravu vnitřních dat o dané entitě (viz Obrázek 28). Každá entita má tuto úpravu určenou svým předkem a díky polymorfismu dochází při úpravě dat k zapouzdření. Objekt dané entity si určí jak a která data zpřístupní grafickému uživatelskému rozhraní pro změnu a potom danou úpravu i sám obslouží. U vazeb můžeme měnit orientaci, u tříd potom jejich název a metody/atributy.
Jednotlivé kreslící panely jsou většinou v aplikaci použity ještě s dalšími komponentami.

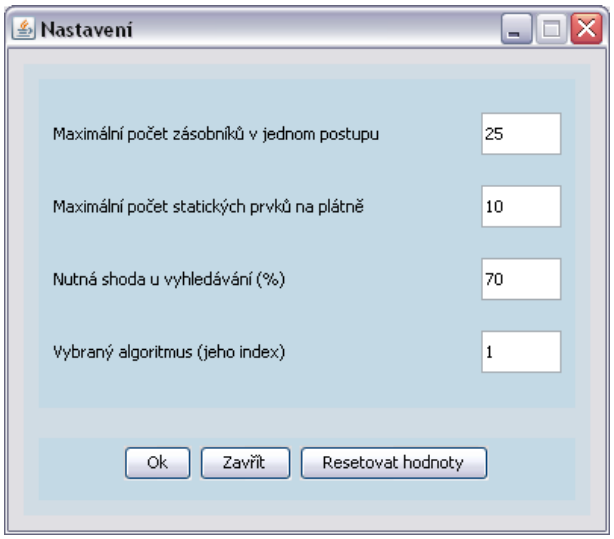
U vyhledávání jsou k dispozici ještě dvě tlačítka, jedno na vyčištění celé pracovní plochy, druhé na samotné vyhledávání. Kroky jednotlivých postupů potom obsahují textové pole pro dodatečný slovní popis kresby a tlačítko na uložení právě tohoto kroku (abychom nemuseli ukládat všechna data).

9.3 Nastavení

Možnost nějak nastavit aplikaci výrazně ovlivňuje funkčnost aplikace a také přispívá k uživatelskému komfortu. Tento komfort je vyjádřen především tím, že aplikace si sama ukládá a při spuštění načítá svou poslední velikost a umístění. Aplikace si také pamatuje naposledy vybraný algoritmus, uživatel tedy nastavuje tuto preferenci pouze jednou a nemusí ji při každém spuštění znova měnit.
Dalším možným nastavením je stanovení maximálního počtu zásobníků v jednom postupu, což lze chápat jako maximální počet jednotlivých kroků pro řešení daného problému. Pro aplikaci toto omezení není až tak podstatné, ale výrazně přispívá k přehlednosti všech řešení. Určitým způsobem nutí uživatele rozebrat problém v těchto daných krocích a nepoužít jich zbytečně více.
Parametr maximálního počtu statických prvků výrazně ovlivňuje rychlost vyhledávání, resp. čím více statických prvků bude na plátně, tím větší bude graf a následné porovnávání takovýchto grafů bude časově složitější. Pokud chceme v budoucnu výrazně zrychlit vyhledávání, možným řešením je právě omezení velikosti výsledných grafů.
Dalším hlediskem, které musíme uživateli dovolit nastavit, je minimální požadovaná shoda dvou grafů, aby byly grafy vyhodnoceny jako podobné. I když tímto nastavením vyhledávání neurychlíme (stejně musíme nalézt největší možný společný podgraf), můžeme korigovat množství nalezených výsledků a zvýšit tak jejich přehlednost.
Soubor ve formátu txt, který aplikace využívá vypadá takto:

```
Nutná shoda u vyhledávání (%):70
Sirka:1032
Pozice y:-4
Pozice x:-4
Vybraný algoritmus (jeho index):0
Maximální počet statických prvků na plátně:10
Maximální počet zásobníků v jednom postupu:25
Výška:574
```

Při spuštění aplikace je vytvořen objekt Setup, který data z tohoto souboru získá, poskytuje k nim přístup a další metody pro jejich správu. Třída Setup využívá návrhový vzor Jedináček a aplikace tedy smí využívat pouze jeden Setup objekt a jeden příslušný soubor. Data pro nastavení jsou uložena v souboru v textové formě z důvodu externího a poměrně přehledného nastavování aplikace, bez toho aniž bychom aplikaci vůbec museli spouštět. Použití souboru sebou nese ovšem několik rizik a je nutné je předem ošetřit. Těmito riziky je celá řada výjimek, které mohou nastat při čtení, otvírání, zavírání souboru a neplatnosti předkládaných hodnot (např. položka pozice obsahuje písmeno). Příkladem takové chyby může být absence Setup souboru. Objekt Setup obsahuje tabulku výchozích hodnot, které jsou v takovém případě použity místo chybných položek. V tomto případě tedy budou všechny položky vyhodnoceny jako chybné a nahrazeny výchozími hodnotami, které budou při ukončení aplikace uloženy do nově vytvořeného souboru.
Přímo v aplikaci lze tento soubor spravovat prostřednictvím položky Nastavení v menu (viz Obrázek 29). Výsledné okno je vytvářeno dynamicky na základě tabulky všech dostupných a aplikačně nastavitelných položek (mezi takovéto položky nepatří třeba velikost a pozice otevřeného okna) v objektu Setup. Díky této dynamičnosti je budoucí správa takového okna velmi usnadněna.



Obrázek 29: Okno nastavení aplikace

9.4 Správa postupů

Funkčnost aplikace vyžaduje možnost spravovat postupy v databázi prostřednictvím grafického rozhraní aplikace. Součástí takové správy je formulář pro vytváření postupů. Ten uživateli umožňuje jednoduše vytvořit celý postup o n-krocích a daném názvu. Posledním krokem vytvořeného postupu by mělo být pojmenované řešení.
Přehled všech dat (tedy postupů a jejich zásobníků) je umístěn přehledně vedle obsahu v Java komponentě JTree. Na této komponentě je zachytáváno kliknutí pravým tlačítkem myši na kteroukoliv položku. Odpovědí na tuto akci je zobrazení potomka komponenty JPopupMenu (vyskakovací okno). Toto okno poskytuje přehled všech dostupných operací nad postupem nebo jeho krokem. Požadované operace už byli nastíněny v návrhu, zde tedy bude zmíněna jejich následná realizace.

- přejmenování kroku nebo celého postupu – Na tento požadavek je uživatel v novém okně požádán o zadání nového jména. Toto jméno je potom změněno v příslušném objektu entity.

- smazání kroku nebo celého postupu – Při operaci smazání je nutné smazat data nejen z vnitřního načteného datového modelu, ale i v otevřené databázi.

- přidání nového zásobníku (kroku postupu) – Při této akci je uživatel požádán o jméno nového kroku. Následně je vytvořen nový krok, jeho kreslící panel, a ten je umístěn v přehledu i v hierarchii o jeden blok před označený zásobník ⇒ nikdy nelze přidat nový krok na konec celého postupu.

- kopírování kroků – Kopírování dvou kroků mezi sebou je poměrně složitá operace. Pro její realizaci je využito rozhraní cloneable jazyka Java. Všechny entity (a jejich příslušné specifikace–metody tříd, atd.), tedy včetně vazeb, jsou překopírovány, jedna po druhé, až po nejnižšího potomka v celé hierarchii, do určeného postupu. Protože vazby obsahují reference na spojované statické prvky a statické prvky na své vazby, nemůžeme reference ve vazbách také klonovat z důvodu zacyklení celé operace. Reference v těchto vazbách jsou proto nahrazeny aktuálními až po procesu klonování.

```
1 public Object clone() throws CloneNotSupportedException{
2         Object obj = super.clone(); //předek
```

```
3           ArrayList<Vazba> vazby_kopie = new ArrayList<Vazba>(); //kopie vazeb
4           int velikost = vazby.size();
5           for (int i = 0; i < velikost; i++) {
6                   //vazba implementuje cloneable a vrací svůj klon
7               vazby_kopie.add((Vazba)vazby.get(i).clone());
8           }
9           ((StatickePrvky) obj).setVazby(vazby_kopie); //nastavení nových hodnot
10          return obj;
11      }
```

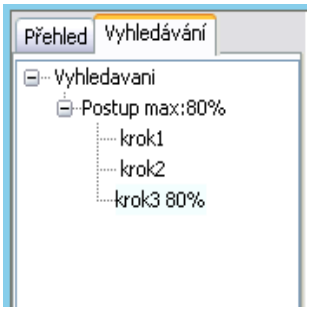
Po všech zmíněných operacích je nutné provést aktualizaci zobrazeného přehledu dat i obsahu. Přehledy zobrazených dat nemají vlastní kontrolor a žádají data o provedení všech výše zmíněných operací. Je tedy nutné vytvořit synchronizační třídu mezi několika přehledy. Každý přehled si udržuje synchronizační objekt, který obsahuje seznam ostatních přehledů a nutné synchronizační operace.

```
1 //provážeme pomocí synchronizace tyto dva panely
2 prehled.getSynchronizace().pridejSynchronizace(vyhledavani);
3 vyhledavani.getSynchronizace().pridejSynchronizace(prehled);
```

Operace na datech v přehledu lze realizovat také pomocí položky akce v menu aplikace. Zde jsou zobrazeny všechny dostupné akce, které aplikace umožňuje. Pokud se pokusíme provést operaci nad položkou, která ji nepodporuje, budeme upozorněny a operace se neprovede. Příkladem může být snaha přidat nový postup tímto způsobem.

9.5 Vyhledávání

Vyhledávání je základní částí aplikace, a jak už bylo nastíněno ve zvláštní kapitole, jedná se o velmi složitou a komplexní činnost využívající algoritmus, který může v nejhorším případě dosahovat dlouhé časové náročnosti.
Zde se již nebudu k těmto algoritmům a jejich pseudokódům vracet. Na místo toho tu ukáži realizaci vyhledávání z pohledu grafického rozhraní a možností uživatele.
Tlačítko pro vyhledávání je umístěno pod vyhledávacím kreslícím panelem, umístěným v první záložce obsahu. Vyhledávací kreslící panel obsahuje vlastní kreslící algoritmus, který se momentálně liší převážné v přístupu k datům v datové části MVC. Diagram nakreslený na tento panel je považován za **hledaný problém** a pomocí grafové teorie je porovnáván se všemi diagramy ve všech krocích všech postupů.
Protože vyhledávání je časově náročná operace, je uživateli zobrazena komponenta ProgressMonitor, kde lze vidět kolik diagramů už bylo zpracováno a kolik jich ještě zbývá.
Tato komponenta obsahuje i metodu progressMonitor.isCanceled(), která je dále distribuována do samotného algoritmu a dovoluje uživateli v případě dlouhého porovnávání ukončit algoritmus během jeho výpočtů.
Výsledný podgraf je potom zprůměrován s maximálními možnými hodnotami. Tím rozumíme počet hran E_c podgrafu C s hranami E_1, pokud platí, že počet hran $E_1 > E_2$ a počet vrcholů V_c s vrcholy V_1, pro které platí, že počet vrcholů $V_1 > V_2$, kde $G_1 = (V_1, E_1)$ a $G_2 = (V_2, E_2)$. Výsledkem je průměrná hodnota shody mezi dvěma grafy. Ta je porovnána s nutnou shodou nastavenou v nastavení a pokud je vyšší nebo rovna, potom je příslušný postup zobrazen na vyhledávací panel.
Vyhledávací panel je umístěn ve stejném panelu jako přehled dat (tato část je přepínací) a s tímto panelem je synchronizován $\Rightarrow$ operace provedené ve vyhledávání se promítnou i na druhý přehled. Výsledky vyhledávání jsou zobrazeny jako postupy s nejlepší dosaženou shodou. U každého kroku je potom vypsána jeho shoda (viz Obrázek 30).
Abychom mohli tyto hodnoty vypsat, musíme upravit vykreslování stromu přepsáním příslušné metody.



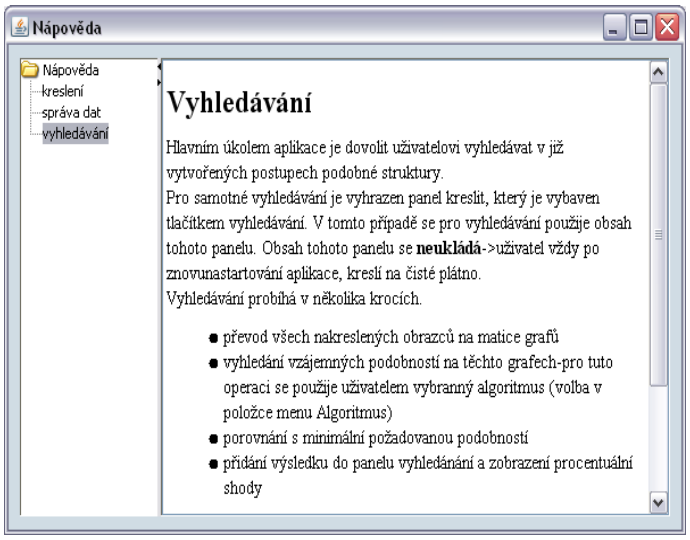
Obrázek 30: Přehled s vyhledáváním

```
1 public Component getTreeCellRendererComponent(JTree tree, Object value,
2             boolean selected, boolean expanded, boolean leaf, int row,
3             boolean hasFocus) {}
```

Výběr použitého algoritmu lze realizovat přes menu bud'to položkou Algoritmus a příslušnou změnou, nebo v položce Nastavení napsat index algoritmu. Index odpovídá zobrazenému pořadí algoritmů v položce Algoritmus. Pokud takto zadáme index algoritmu, který aplikace nezná, bude použit základní algoritmus a zvolené nastavení bude ignorováno.

9.6 Nápověda

Dostupnost nápovědy je snahou zpříjemnit a osvětlit novému uživateli začátky s mnou navrženou aplikací. Nápověda je skryta v položce menu Nápověda. Po vybrání této položky se uživateli zobrazí nové okno se seznamem témat (viz Obrázek 31). Po vybrání tématu se načte jeho popis z příslušného htm souboru složky nápověda.



Obrázek 31: Nápověda

Témata i obsah nápovědy v aplikaci jsou generovány dynamicky na základě obsahu této složky. V potaz se berou pouze htm soubory, a proto je před samotným načtením dat obsah složky filtrován.

```
1 File[] temata = slozka_napoveda.listFiles(new FileFilter() { //načteme témata ze složky
2         @Override
3   public boolean accept(File pathname) { //použití filteru
4         if(pathname.getName().toLowerCase().endsWith("htm")) { //pouze htm soubory
5         return true;
```

```
6       } else {
7         return false;
8       }
9     }
10 });
```

Toto dynamické chování přispívá k její budoucí snadné editaci a rozšířitelnosti. Soubory htm jsou zde použity z důvodu lepší vizuálnosti a strukturovanosti textu. Komponenta JEditorPane jazyka Java nám dovoluje vykreslit text na základě použitých htm značek.

10 Závěr

Bakalářská práce měla za úkol představit několik návrhových vzorů a MVC architekturu jako jeden ze způsobů návrhu struktury softwaru v závislosti na definovaném problému. Popsat a určit další možné způsoby analýzy problému z pohledu jejich možné automatizace a zohlednit jejich použití v navrhovaném systému.
Po zvážení všech možných řešení, byl vybrán pro tuto analýzu standardizovaný jazyk UML a jeho diagram tříd. Tento diagram byl vybrán z důvodů srozumitelnosti, menšího množství informací v textové podobě a poměrně velkého množství přenášených dat.
Aby uživatel mohl na základě takto definovaného problému vybrat optimálnější řešení, bylo navrženo grafické rozhraní pro správu postupů a jednotlivých kroků řešení, které uživatel už v minulosti použil.
Navržený systém poskytuje grafické rozhraní pro zadávání resp. nakreslení diagramu tříd. Problém definovaný tímto diagramem je následně porovnán s uloženými daty a na základě shody s již vyřešenými projekty je vybrán postup, který takovýto problém už zdárně vyřešil. Uživatel může pokračovat po jednotlivých krocích vybraného postupu až k danému řešení.
Nejtěžším problémem realizace takového systému se ukázalo samotné porovnávání dvou grafů, které symbolizují příslušné diagramy. Tento problém byl za pomoci grafové teorie definován jako hledání maximálního společného podgrafu. Bakalářská práce popisuje vhodné algoritmy a principy řešení takového problému.
Návrh systému byl uskutečněn s přihlédnutím k budoucímu rozšiřování aplikace o mnoho dalších aspektů (nové kreslené prvky, diagramy, algoritmy, rozšiřování nápovědy, nastavení, atd.). V jednotlivých krocích návrhu jsou popsány požadavky kladené na aplikaci, použití návrhových vzorů nebo principů objektově orientovaného programování k jejich vyřešení a z toho plynoucí výhody.
Analýza problému a následné automatizované zpracování se ukázalo jako velmi komplexní problém. Aplikace a bakalářská práce nastiňují možné řešení, ovšem výsledné použití pouze jednoho diagramu zvyšuje nepřesnost nalezených výsledků. Diagram tříd jazyka UML převedený na graf nedokáže přenést dostatečné množství informací k exaktnímu definování problému a řešení. Tohoto nedostatku jsem si byl vědom, a proto jsem při vývoji aplikace počítal s možností rozšíření, které spočívá v přidání dalších diagramů, což zvýší množství poskytovaných informací o problému a zpřesní vyhledávání. V budoucnosti je dále možné aplikaci vylepšit zvýšením počtu využívaných algoritmů.

Použitá literatura

[1] DVOŘÁK, Miloš, et al. Objekty – Vzory [online]. 27.05.2005, 03.02.2008 [cit. 2011-04-12]. Dostupný z WWW: <http://objekty.vse.cz/>.

[2] BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. The unified modeling language user guide. 2nd ed. Upper Saddle River : Addison–Wesley, 2005. 475 s. ISBN 0321267974.

[3] KANISOVÁ, Hana; MÜLLER, Miroslav. UML srozumitelně. 2. aktualiz. vyd. Brno : Computer Press, 2007. 176 s. ISBN 8025110834.

[4] GAMMA, Erich. Design patterns elements of reusable object–oriented software. Reading : Addison–Wesley, 1995. 395 s. ISBN 0201633612.

[5] VLISSIDES, John. Pattern hatching : design pattens applied. Reading, Mass. : Addison–Wesley, 1998. 172 s. ISBN 020143293.

[6] Head first design patterns. Sebastopol, CA : O'Reilly, 2004. 638 s. ISBN 0596007124.

[7] YACOUB, Sherif M; AMMAR, Hany H. Pattern-oriented analysis and design : composing patterns to design software systems. Boston : Addison-Wesley, 2004. 385 s. ISBN 0201776405.

[8] BUSCHMANN, Frank. Pattern-oriented software architecture : a system of patterns. Chichester: John Wiley & Sons, 1996. ISBN 0471958697.

[9] KERIEVSKY, Joshua. Refactoring to patterns. Boston : Addison-Wesley, 2005. 367 s. ISBN 0321213351.

[10] LARMAN, Craig. Applying UML and patterns : an introduction to object-oriented analysis and design. Upper Saddle River : Prentice Hall PTR, 1998. 507 s. ISBN 0137488807.

[11] VALIENTE, Gabriel. Algorithms on trees and graphs. Berlin : Springer, 2002. 490 s. ISBN 3540435506.

[12] RAYMOND, J., WILLETT, P. Maximum common subgraph isomorphism algorithms for the matching of chemical structures [online]. Journal of Computer-Aided Molecular Design, 2002, vol. 16, str. 521-533. ISSN 1573-4951. Dostupný na URL: <http://eprints.whiterose.ac.uk/3569/1/willetts4.pdf>.

[13] ČECHOVÁ, Andrea. Porovnávání proteinů reprezentovaných obecným grafem. Brno, 2010. Bakalářská práce. Masarykova univerzita Fakulta informatiky.

[14] CONTE, Donatello, Pasquale FOGGIA a Mario VENTO. Challenging Complexity of Maximum Common Subgraph Detection Algorithms: A Performance Analysis of Three Algorithms on a Wide Database of Graphs. Journal of Graph Algorithms and Applications. 2007, vol. 11. Dostupné z: <http://ftp.findthatfile.com/search-14569357-fPDF/download-documents-contefoggiavento2007.11.1.pdf.htm>

[15] MCGREGOR, James J. Backtrack Search Algorithms and the Maximal Common Subgraph Problem. Software-Practice and Experience. 1982, č. 12.

[16] WANG, Yu a Carsten MAPLE. A Novel Efficient Algorithm for Determining Maximum Common Subraphs. Department of Computing and Information Systems, University of Luton, UK. 2005.

[17] ZAGER, Laura a George VERGHESE. Graph similarity. EECS, MIT [online]. [cit. 2012-03-13]. Dostupné z: <http://lees-web.mit.edu/lees/presentations/LauraZager.pdf>

[18] REGNERI, Michaela. Finding All Cliques of an Undirected Graph. Seminar: Current Trends in IE. 2007

[19] CAZALS, F. a C. KARANDE. A note on the problem of reporting maximal cliques. 2008. Dostupné z: <ftp://ftp-sop.inria.fr/abs/fcazals/papers/ncliques.pdf>

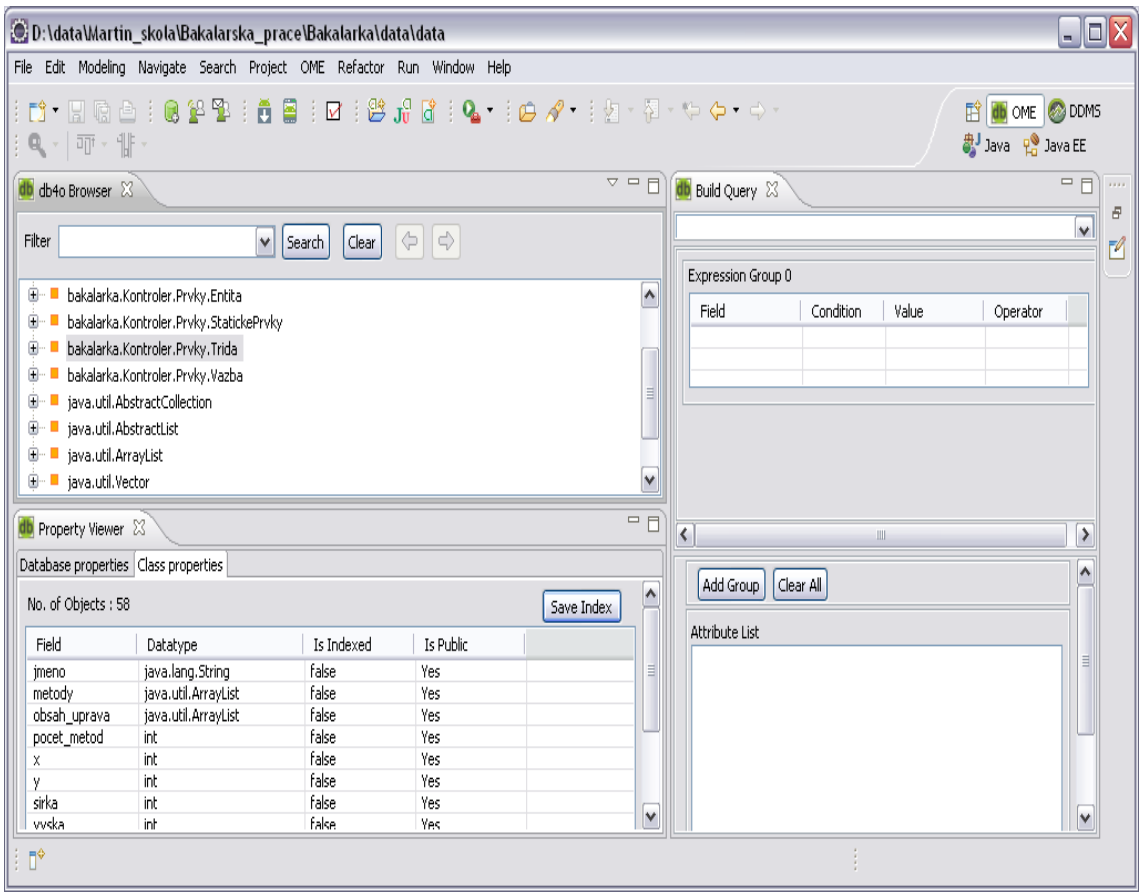
[20] Db4o. Db4objects [online]. 2005, 2011 [cit. 2012-03-13]. Dostupné z: <http://www.db4o.com>

[21] Community Material. Hibernate [online]. 2012 [cit. 2012-03-13]. Dostupné z: <http://www.hibernate.org/docs>

[22] Creating a GUI With JFC/Swing. The Java Tutorials [online]. 2012 [cit. 2012-03-13]. Dostupné z: <http://docs.oracle.com/javase/tutorial/uiswing/index.html>

Příloha A

Tato část zobrazuje otevřenou databázi db4o v programu Eclipse s použitým pluginem.



Obrázek 32: Program Eclipse a databáze db4o

Příloha B

Přiložené CD obsahuje:

- zdrojové kódy a všechny podpůrné soubory aplikace
- .jar knihovnu databáze db4o
- spustitelný soubor systému + ukázku vytvořené databáze v db4o
- zdrojové kódy použitých algoritmů
- bakalářskou práci ve formátu .pdf a .tex
- použité obrázky